Synthesis of Irregular Combinational Functions with Large Don't Care Sets

V. Gherman
CEA, LIST, Boîte Courrier 65
Gif-sur-Yvette, F-91191 France
valentin.gherman@cea.fr

H.-J. Wunderlich, R. Mascarenhas
Universitaet Stuttgart, Pfaffenwaldring 47
Stuttgart, D-70569 Germany
{wu, masca}@ra.informatik.uni-stuttgart.de

J. Schloeffel, M. Garbers
NXP Semiconductors, Georg-Heyken-Str. 1
Hamburg, D-21147 Germany
{juergen.schloeffel, michael.garbers}@nxp.com

ABSTRACT

A special logic synthesis problem is considered for Boolean functions which have large don't care sets and are irregular. Here, a function is considered as irregular if the input assignments mapped to specified values ('1' or '0') are randomly spread over the definition space. Such functions can be encountered in the field of design for test. The proposed method uses *ordered* BDDs for logic manipulations and generates *free* BDD-like covers. For the considered benchmark functions, implementations were found with a significant reduction of the node/gate count as compared to SIS or to methods offered by a state-of-the-art BDD package.

Categories and Subject Descriptors
B.6.3 [**Logic Design**] Design Aids.

General Terms: Algorithms, Design.

Keywords: logic synthesis, incompletely specified functions.

1. INTRODUCTION

Incompletely specified Boolean functions are characterized by a non-empty *don't care* (DC)-set, which may become very helpful in optimizing the logic implementation of these functions. Two-level and multi-level logic implementations have been extensively investigated and the outcome is a couple of programs like ESPRESSO [4], MIS [3], SIS [22], the multi-level logic synthesizer implemented by Minato [16] and tools from private companies like Synopsys, Cadence, etc. These tools work quite well for regular problems or for small and medium sized irregular problems. Unfortunately, they are not so efficient for large problem instances in which the input assignments mapped to '1' or '0' are randomly distributed over the definition space. Such *irregular functions* may appear in areas like design for test (DFT) [10][25][26] and artificial intelligence [15].

In MIS, the DC-based optimization relies on ESPRESSO or simpler variants of it [3], which can act only on the two-level representation of the functions implemented by each node of a Boolean network. This does not necessarily guarantee a reduction of the size of the Boolean network [3]. In [16], first a BDD-based representation is generated for the target circuit and for its DC-set. Minato-Morreale's algorithm [16] is used to obtain a prime-irredundant cube cover out of the BDD-based representation. Finally, the cube cover is transformed into a multi-level circuit by using a heuristic for fast algebraic-division. However, the DC-set is only used for optimizations of two-level representations in both multi-level synthesis approaches.

GLSVLSI'07, March 11–13, 2007, Stresa-Lago Maggiore, Italy.

A minimal BDD can offer a good starting point for a multi-level implementation. If each node of a BDD [1][5] is substituted by a multiplexer, a multi-level circuit can be easily generated [2]. The minimization of an ordered BDD (OBDD)-based implementation using the DC-set is NP-hard [20]. In [17], an exact OBDD minimization algorithm based on the DC-set is presented. Due to the NP-hardness of the problem, this approach has a limited applicability.

Some of the first OBDD minimization heuristics that take advantage of the DC-set are the operators *constrain* and *restrict* [8][9]. Other OBDD minimization methods exploit the DC-set for sibling matching or, more generally, for matching BDD nodes below cut lines through the BDD, which enables a more aggressive BDD reduction [7][23]. None of these two methods is safe, which would require that the resulting BDD is always smaller than the original one. The compaction algorithm of [13] avoids this problem by using a preprocessing step to identify the nodes that can make the minimization unsafe. Compared to *restrict* or *constrain* this compaction algorithm gives better results on the average, but it is considerably slower. Moreover, none of the heuristics analyzed in [23] succeeds to outperform *restrict* by more than a few percents. In [21], the concept of variable reordering based on symmetries has been extended to incompletely specified functions such that an OBDD can be minimized by means of *don't care* assignments combined with variable reordering. This method cannot handle large problem instances.

Free BDDs (FBDDs) [11][24] are more compact than OBDDs, due to the fact that in the former case the order of variables on the paths from the root-node to a leaf-node is not fixed. There are functions, e.g. the hidden weighted bit function, which require OBDD-based representations of exponential size [6], while FBDD-based representations of polynomial size are known [24]. In the case of completely specified functions, exact and heuristic algorithms for the minimization of FBDD-based representations are described in [11]. Unfortunately, despite sophisticated pruning techniques, the exact approach is inherently bound to very small problems (with a maximum of 8 input variables).

This paper proposes the first FBDD-based logic synthesis method for incompletely specified functions. In the case of irregular functions with large DC-sets, the method improves considerably all the synthesis parameters as compared to SIS or OBDD-based approaches.

The next section presents two examples of incompletely specified Boolean functions that are irregular and have large DC-sets. Section 3 presents our heuristic method to find efficient implementations for such functions. In Section 4, experimental results are used to compare the new approach with SIS [22] and methods available in the CUDD-package [27] (*restrict* [9]). Furthermore, the outcome of the new method is evaluated as input to Synopsys Design Compiler. The paper is summarized in Section 5.

2. EXAMPLES OF IRREGULAR BOOLEAN FUNCTIONS WITH LARGE DC-SETS

Deterministic logic built-in self-test (DLBIST) is a DFT technique utilized to test logic cores. This technique uses a pseudo-random sequence (produced by an LFSR, for example) to test the majority of faults of the core under test (CUT). Test responses are usually compressed by a multi-input shift register (MISR) [10][14][19][26].

In order to detect more faults, deterministic cubes are embedded into the pseudo-random test sequence. In the *bit-flipping* DLBIST approach [10], the modification of the pseudo-random patterns is realized by inverting *(flipping)* some of the LFSR outputs, such that deterministic patterns are obtained (Figure 1). The bit-flipping is performed by combinational logic implementing a so-called bit-flipping function (BFF).

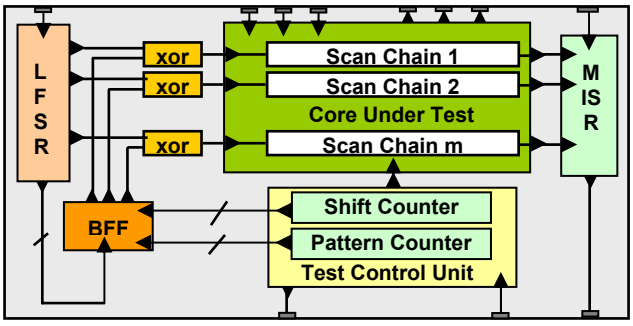


Figure 1: Bit-flipping DLBIST architecture.

The test responses may contain unknown bits (Xs), which can appear due to the existence of multiple clock domains, floating buses or non-initialized memory elements. In order to obtain an uncorrupted signature at the end of the test, these Xs have to be *masked* to either logic '0' or logic '1' before they propagate into the MISR (Figure 2). This may be performed by combinational logic implementing a so-called *X-masking function* (XMF) [25].

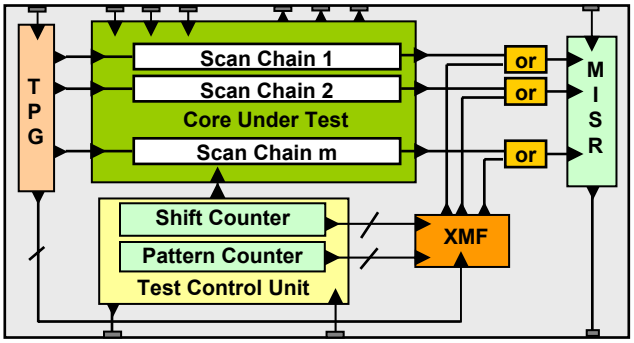


Figure 2: Embedded test architecture with MISR.

The inputs of the BFF and the XMF are the state bits of the pattern counter, the shift counter and the test pattern generator (TPG) which can be an LFSR. Both functions are incompletely specified. In the BFF case, the ON-set includes the states that correspond to the clock cycles in which the LFSR output must be flipped. Similarly, the OFF-set contains the states that correspond to the clock cycles in which the LFSR output must not be flipped. The DC-set includes the states that correspond to the clock cycles in which this output may be arbitrarily flipped.

In the XMF case, the ON-set contains the states that correspond to the clock cycles in which an unknown test response bit must be masked before it is scanned into the MISR. Similarly, the OFF-set includes the states that correspond to the clock cycles in which a test response bit carrying the information about the CUT correctness must not be masked. The DC-set contains the states that correspond to the clock cycles in which the test response bits may be arbitrarily masked before they are propagated into the MISR.

The DC-sets cover more than 99.99% of the definition space of both functions while the ON-sets and the OFF-sets are randomly distributed over the rest of the definition space. One can identify the following sources of the high cardinality of the DC-sets.

- Not all the possible states and state combinations of the Shift Counter, Pattern Counter and LFSR (Figure 1) are necessarily appearing during the testing process.
- In the case of the BFF, the deterministic test cubes that have to be mapped to the pseudo-random test sequence contain many don't care bits and the number of embedded deterministic test cubes is a small fraction of the total number of pseudo-random test patterns.
- In the case of the XMF, usually a very small fraction of the bits in test responses are Xs or relevant to the fault coverage.

The large DC-sets offer a good base for the optimization of the logic implementation of these functions despite their irregularity, which is not the case with random functions with no or small DC-sets.

3. PROPOSED FBDD-BASED LOGIC SYNTHESIS

Below, an incompletely specified function $F:\{0,1\}^N \rightarrow \{0,1,-\}$ (the symbol '-' indicates a *don't care*) will be represented by the characteristic functions f_{on}, and $f_{off}:\{0,1\}^N \rightarrow \{0,1\}$ of its ON-set and OFF-set, respectively. This representation of F will be denoted by $F(f_{on}, f_{off})$.

A function $Cov(F):\{0,1\}^N \rightarrow \{0,1\}$ will be called a *cover* of $F(f_{on}, f_{off})$ iff: $f_{on} \cdot Cov(F) = f_{on}$ and $f_{off} \cdot Cov(F) = 0$. The cofactor of a function f by a literal $l \in \{x, \neg x\}$ will be denoted by $f|_l$, where x is an input variable of f. The number of fully specified input assignments $\{x \in \{0,1\}^N | f(x)=1\}$ will be denoted by $||f||$ (*cardinality* of f).

The goal of the synthesis procedure described here is to generate FBDD-like covers with a reduced gate count in the resulting circuit descriptions. This is achieved by first reducing the number of paths from the root node to a leaf node and second by looking for node sharing among different paths and even different FBDDs.

Each path in a BDD corresponds to a sub-space which is mapped either to 1 or to 0. Similarly, the cover of the function $F(f_{on}, f_{off})$ can be chosen equal to 0 on the subspaces mapped by f_{on} to 0 and equal to 1 on the subspaces mapped by f_{off} to 0. Consequently, the path reduction of the FBDD-based implementation can be achieved by finding a minimal partition of the definition space of the considered function into appropriate sub-spaces on which either f_{on} or f_{off} is equal to 0. Given the function $F(f_{on}, f_{off})$ and the set of its input variables V, the synthesis method introduced here looks for a *good* partition of the definition space into such special sub-spaces using the recursive depth-first process sketched below.

First, it is decided whether $F(f_{on}, f_{off})$ or $\neg F(f_{off}, f_{on})$ is implemented, depending on the compactness of the OBDD-based representation of f_{on} and f_{off}. The OBDD sizes are determined by their node count. Subsequently, a variable x is determined (in *CreateLiteralCover* or *SplitOperator*) with respect to which the current definition subspace is decomposed into 2 new subspaces where x is either 1 or 0. For each of the 2 subspaces a further

recursive call of *CreateCover* may be required. The size of the resulting cover may be reduced by determining a minimal number of such successive recursive calls. *CreateLiteralCover* and *Split Operator* implement heuristics to obtain near-optimal solutions.

CreateCover (f_{on}, f_{off}, V){
  if size(f_{on}) > size(f_{off}) **then return** ¬**CreateCover** (f_{off}, f_{on}, V);
  $l = \varnothing$;
  Cov = **CreateLiteralCover** (f_{on}, f_{off}, V, l);
  if ($Cov \neq \varnothing$) **then return** Cov;
  Cov = **FindCover** (f_{on}, f_{off}); // optional DC-based node reduction
  if ($Cov \neq \varnothing$) **then return** Cov;
  for all $i \in V$ **and for** $l_i \in \{x_i, \neg x_i\}$
    if ($f_{on}|_{li} = 0$ **and** $f_{off}|_{li} = 0$) **or**
    ($f_{on}|_{li} = f_{on}|_{\neg li}$ **and** $f_{off}|_{li} = f_{off}|_{\neg li}$) **then** $V = V - \{i\}$;
  if $l \neq \varnothing$ **then**
    Cov = **CreateCover** ($f_{on}|_{\neg l}, f_{off}|_{\neg l}, V$);
    if $f_{off} \cdot Cov = 0$ **then return** Cov;
    else return $(\neg l) \cdot Cov$; // new FBDD-node required
  return SplitOperator ($f_{on}|_{\neg l}, f_{off}|_{\neg l}, V$);}

CreateLiteralCover provides the recursive process with the first stop condition. The recursion is stopped if a literal l is found for which $f_{off}|_{\neg l}$ and $f_{on}|_l$ are equal to 0. In this case $\neg l$ is chosen as a cover for F. If this condition cannot be fulfilled and there are literals l_i, for which $f_{on}|_{li}$ is equal to 0, then that literal l_i which minimizes the cardinality of $f_{off}|_{\neg li}$ will be assigned to the generic argument l. *FindCover* which provides the algorithm with the second stop condition is optional and will be discussed later.

CreateLiteralCover (f_{on}, f_{off}, V, l){
  $Min = \infty$;
  for all $i \in V$ **and for** $l_i \in \{x_i, \neg x_i\}$
    if $f_{on}|_{li} = 0$ **and** $\| f_{off}|_{\neg li} \| < Min$ **then** $Min = \| f_{off}|_{\neg li} \|$; $l = l_i$;
  if $Min \neq \infty$ **and** $f_{off}|_{\neg l} = 0$ **then return** $\neg l$;
  return $\varnothing$;}

Subsequently, the set of input variables V is pruned from those variables on which f_{on} and f_{off} depend in a *trivial* way (*for* loop of *CreateCover*). Depending on whether the literal l returned by *CreateLiteralCover* is different from the empty set $\varnothing$, either *CreateCover* or *SplitOperator* is called.

Procedure *SplitOperator* uses two heuristics. The first one looks for a literal l such that the cardinalities $\|f_{on}|_l\|$ and $\|f_{off}|_{\neg l}\|$ are higher than the cardinalities $\|f_{off}|_l\|$ and $\|f_{on}|_{\neg l}\|$, respectively. If such an *unbalancing* occurs, then the following relation must hold:

$$| \|f_{on}|_l\| - \|f_{off}|_l\| | + | \|f_{off}|_{\neg l}\| - \|f_{on}|_{\neg l}\| | > | \|f_{off}\| - \|f_{on}\| | \qquad (1)$$

The intuition behind the unbalancing is that we heuristically try to find the literal l that simultaneously minimizes both cardinalities $\|f_{on}|_l\|$ and $\|f_{off}|_{\neg l}\|$. For example, consider the definition space presented in Figure 3, where the symbols 'x' and 'o' are used to represent the input assignments belonging to the ON-set and the OFF-set of the considered function, respectively. The dashed squares give a minimal partition of the definition space into sub-spaces containing only input assignments belonging either to the ON-set or to the OFF-set. Assume that one has to choose between the input variables x_1 and x_2 for the decomposition of the considered definition space. The other input variables are not explicitly shown for simplicity reasons. The enclosed table shows the number of input assignments belonging to the ON-set and the OFF-set in the sub-spaces defined by $x_1 = 1$, $x_1 = 0$, $x_2 = 1$ and $x_2 = 0$. In this case, the first heuristics of the procedure *SplitOperator* chooses the variable x_1 with respect to which the definition space is *unbalanced* and the inequality (1) is fulfilled. The left-hand side member of the inequality (1) is evaluated to 15/3 with respect to the variable x_1/x_2. In total, there are 13/10 input assignments belonging to the ON-set/OFF-set, so that the right-hand side member of the inequality (1) is evaluated to 3. It can also be observed that the cut line corresponding to the decomposition of the definition space with respect the input variables x_1 does not intersect any sub-space of the minimal partition. This does not happen in the case of the variable x_2.

SplitOperator (f_{on}, f_{off}, V){
  $Max = 0$;
  for all $i \in V$ // first heuristic
    $Check = | \|f_{on}|_{xi}\| - \|f_{off}|_{xi}\| | + | \|f_{off}|_{\neg xi}\| - \|f_{on}|_{\neg xi}\| |$;
    if $Check > Max$ **then** $Max = Check$; $m = i$;
  if $Max = | \|f_{off}\| - \|f_{on}\| |$ **then** // second heuristic
    $MinOn = \infty$; $MinOff = \infty$;
    for all $i \in V$ **and for** $l_i \in \{x_i, \neg x_i\}$
      if $\|f_{on}|_{li}\| < MinOn$ **or**
      if $\|f_{on}|_{li}\| = MinOn$ **and** $\|f_{off}|_{\neg li}\| < MinOff$ **then**
        $MinOn = \|f_{on}|_{li}\|$; $MinOff = \|f_{off}|_{\neg li}\|$; $m = i$;
  $V = V - \{m\}$; // choose the literal for the first recursion
  choose $l \in \{x_m, \neg x_m\}$ **such that** $\|f_{off}|_l\| \geq \|f_{off}|_{\neg l}\|$;
  Cov_1 = **CreateCover** ($f_{on}|_l, f_{off}|_l, V$);
  if $Cov_1 \cdot f_{off} \neq 0$ **then** Cov_2 = **CreateCover** ($f_{on}|_{\neg l}, f_{off}|_{\neg l}, V$);
  else if $f_{off}|_{\neg l} \neq 0$ **then** Cov_2 = **CreateCover**($\neg Cov_1 \cdot f_{on}|_{\neg l}, f_{off}|_{\neg l}, V$);
  else $Cov_2 = \neg 1$;
  if $Cov_1 \cdot f_{off} \neq 0$ **then** $Cov_1 = l \cdot Cov_1$;
  if $Cov_2 \cdot f_{off} \neq 0$ **then** $Cov_2 = \neg l \cdot Cov_2$;
  return $Cov_1 + Cov_2$;}

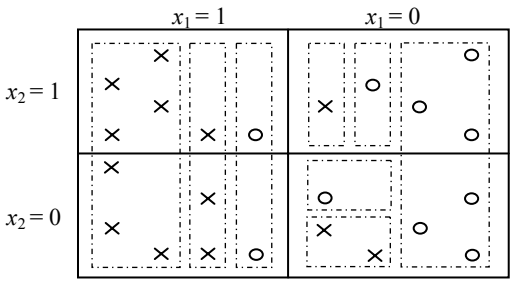
If no *unbalancing* variable has been found, then the second heuristic is used. This heuristic chooses the variable x, which has an associated literal $l \in \{x, \neg x\}$ that minimizes the cardinality $\|f_{on}|_l\|$ as a primary optimization goal and minimizes the cardinality $\|f_{off}|_{\neg l}\|$ as a secondary optimization objective. The first optimization goal is similar to the approach used in [12]. The intuition behind this is similar to the one mentioned for the first heuristic of *Split Operator*. For each literal $l \in \{x, \neg x\}$ a recursive call with the argument ($f_{on}|_l, f_{off}|_l$) is performed iff $f_{off}|_l \neq 0$.

Both heuristics in *SplitOperator* are used to increase the chance of fulfilling the stop condition from *CreateLiteralCover* in the next recursive calls and thus to decrease in a greedy manner the number of subsequent recursive calls of *CreateCover*.

In order to limit the memory consumption of the whole process, the cofactor $f|_x$ is computed using the operator *BDD.Compose* instead of the operator *BDD.And*. In this way, the dependence of the cofactor $f|_x$ on the variable x is eliminated.

The heuristics used here to choose the new variable x depend only on the distribution of the ON-set and of the OFF-set over the definition space of the target function F. This makes the algorithm largely independent of the variable order used for the underlying OBDD-based representation, which is not the case with the heuristic used in [11] for completely specified functions.

A FBDD-based representation is preferred to model the resulting $Cov(F)$, since an OBDD could require excessive memory usage. The FBDD-based representation is constructed node by node during the recursive process. Each non-terminal node of the FBDD is created during a distinct recursion step. The logic implementation of a node created outside *SplitOperator* requires at



	ON-set	OFF-set
$x_1 = 1$	10	2
$x_1 = 0$	3	8
$x_2 = 1$	6	5
$x_2 = 0$	7	5

Figure 3: Example of the proposed decomposition of the definition space.

most one 2-input logic operator, while a node created inside *SplitOperator* may require between one and three 2-input logic operators. *NAND* and *NOR* operators are preferred to *AND* and *OR* operators. In this way the logic is optimized not only by reducing the number of nodes in the FBDD, but also by reducing the operator count per node. Both goals are achieved by exploiting the DC-set.

So far, the node count has been minimized only by attempting to decrease the path count (e.g. looking for minimal partitions of the definition space, where either f_{on} or f_{off} is equal to 0). The node count can be further reduced by allowing non-terminal nodes to become children of more than one parent node and by allowing parent nodes of the same child to belong to FBDDs corresponding to different outputs of the target function. This is nothing else than the well-known node reduction [5] that usually makes the OBDDs very compact, but which in the case of FBDDs is expected to have less impact on the node count.

Procedure *FindCover* is used to check whether the covers *Cov*(*SG*) implemented by already synthesized sub-graphs *SG* are useful also in the case of the target function $F(f_{on}, f_{off})$. If such a sub-graph is found, one has only to point to its root node with a normal or a complemented edge (when $\neg Cov(SG)$ is needed). As long as it is not required that the FBDD-based representations of the resulting covers are canonical, both *else* and *then* edges are allowed to be complemented.

FindCover (f_{on}, f_{off}){
  for each element *SG* of a sub-set of all completed sub-graphs
    if $f_{on} \cdot Cov(SG) = f_{on}$ **and** $f_{off} \cdot Cov(SG) = 0$ **then**
      return *Cov(SG)*;
    if $f_{on} \cdot Cov(SG) = 0$ **and** $f_{off} \cdot Cov(SG) = f_{off}$ **then**
      return $\neg$ *Cov(SG)*;
  return $\varnothing$;}

In order to reduce the node (gate) depth of the cover returned by *CreateCover*, it is important that *CreateLiteralCover* is called before *FindCover*. In order to increase the chances that a cover will be found by *FindCover*, this should be called before the *for-loop* in *CreateCover*. The DC-based node reduction implemented by *FindCover* has the effect that the same node index (variable) may appear more than once on a path going from the root to a terminal node of the resulting cover. Nevertheless, such an effect has never been observed during our experiments, except for some increase of the circuit depth.

Each FBDD node contains a pointer to the OBDD-based representation of the function implemented by its sub-graph. In this way, the Boolean functions involved in the DC-based node reduction can be efficiently manipulated. The run-time and the memory consumption of the search associated with the DC-based node reduction can be reduced by limiting the number of investigated nodes.

The worst case run-time complexity of the FBDD-based logic implementation of an incompletely specified function *F* is proportional to the product of the number of input variables, the maximum size of the OBDD-based representation of each output and the size of the resulting cover. When the DC-based node reduction is enabled, the square of the resulting cover size has to be taken. The node counts of the resulting covers are usually orders of magnitude smaller than the node counts of the original OBDDs.

4. EXPERIMENTAL RESULTS

The FBDD-based approaches published so far do not target the synthesis of incompletely specified functions. Consequently, the proposed FBDD-based method has been evaluated with respect to SIS and the OBDD-based methods available in the CUDD-package [27] able to handle don't cares. The experiments have been performed on GNU Linux machines equipped with 2 GB of memory and an Intel Pentium 4 processor at 2.4 GHz.

Table 1 presents the considered benchmark functions which stem from the field of design for test [10] and can be downloaded from [28]. The 2nd and the 3rd column report the number of inputs and outputs of the target functions. The 4th column (||ON-set|| + ||OFF-set||) gives the sum of the cardinalities of the ON-set and the OFF-set corresponding to each function. The last 2 columns show the (non-terminal) node count of the OBDD-based representation of each function.

Table 2 provides a comparison between the proposed method and OBDD-based approaches with respect to the synthesis of the functions in Table 1. For each approach, we report the number of 2-input logic operators (#gates), the node depth (Node depth) and the 2-input gate depth (Gate depth) of the resulting covers as well as the run-time required to generate these covers (Optimization time). The number of logic operators in the circuit description of a non-terminal FBDD node is obtained by counting the 2-input logic operators in the expression of the corresponding cover Cov(F). In the case of the OBDD-based implementation, the circuit description of each non-terminal node may require 3, 1 or 0 2-input logic operators, depending upon whether the node has 0, 1 or respectively 2 children, that are terminal nodes [2].

Table 1: Multi-output (incompletely specified) target functions.

Multi-output function	#inputs	#outputs	‖ON-set‖ + ‖OFF-set‖	ON-BDD size [#nodes]	OFF-BDD size [#nodes]
p19K	82	24	85,215	615,407	654,443
p59K	77	19	9,918	158,181	315,335
p127K	67	10	663,750	6,876,383	8,067,136

Table 2: Optimization potential of the FBDD-based and the OBDD-based approaches.

Multi-output function	Restrict + Variable Reordering				FBDD			
	#gates	Node depth	Gate depth	Optimization time	#gates	Node depth	Gate depth	Optimization time
p19K	54,672	17	30	0m:20s	8,269	15	24	1m:31s
	39,231	17	30	4m:52s	7,200	23	37	17m:28s
	33,443	17	29	40m:22s	7,161	27	42	22m:07s
p59K	7,084	20	33	2s	1,543	16	25	11s
	4,669	19	31	2m:16s	1,428	23	34	27s
	4,601	19	30	18m:27s	1,423	23	34	1m:10s
p127K	390,057	23	42	24m:21s	120,122	21	36	35m:18s
	256,883	24	42	11h:16m	94,113	68	97	15h:00m
	-	-	-	-	93,837	61	96	16h:34m

Each function has been synthesized three times with each approach. In the case of the FBDD-based approach, the reported experiments show tradeoffs between the run-time and the number of (2-input) logic operators in the circuit description of the resulting covers. These tradeoffs have been obtained by changing the thresholds that control the size of the searching space associated with the DC-based node reduction. The first run corresponding to each function has been done with the DC-based node reduction disabled.

The FBDD-based method has been evaluated with respect to several OBDD-based approaches that use combinations of the restrict operator [9] and variable reordering. The variable reordering has been applied before restrict and to all ON- and OFF-OBDDs corresponding to each output of the target function. As a result, all the covers obtained with the OBDD-based approach have the same variable ordering and, consequently, a maximized probability of node sharing among them. The variable reordering performed on the covers found with restrict for all the outputs of a given function does not bring any node reduction. Consequently, the reported run-time consumption of the OBDD-based approach with variable reordering takes into account only the application of restrict and of the variable reordering done before.

In the first OBDD-based run reported here, no variable reordering has been performed. In the next 2 runs, the variables have been reordered based on the heuristics: CUDD_ REORDER_SYMM_SIFT and CUDD_REORDER_SYMM_ SIFT_CONV [27], respectively. The first heuristic is an implementation of symmetric sifting [18], while the second heuristic is a converging variant of the first one. Variable reordering improves the operator count at the cost of a significant increase in the run-time. The converging heuristic for reordering the variables of the function p127K was still incomplete after days of execution.

The proposed method outperforms all the investigated OBDD-based approaches. Running the FBDD-based flow with the DC-based node reduction switched off results in operator counts (#gates) that are between 2 and 4 times better than those obtained with the best investigated OBDD-based approach. The operator count of the FBDD-based covers can be further improved by enabling the DC-based node reduction and increasing the associated searching space. The FBDD-based approach with DC-based node reduction disabled also provides the implementations with the smallest depths. As mentioned in the previous section, the node depth increases when the DC-based node reduction is enabled, but its maximum is always less than the number of input variables.

The circuit descriptions presented in Table 2 have been synthesized with Synopsys Design Compiler and using a proprietary library. Table 3 reports the resulting area (Cell area) measured in an arbitrary unit, the synthesis run-time (Synthesis time) and the total run-time required to generate the cover (Optimization time, in Table 2) and to synthesize them (Synthesis time, in Table 3).

Compared to the best investigated OBDD-based approach, the FBDD-based flow with the DC-based node reduction disabled reduces area figures by a factor between 2 and 3. This improvement has been achieved by using shorter run-times than for all OBDD-based approaches (if one considers the sum of Optimization time and Synthesis time). Moreover, the run-time of this simple configuration of the FBDD-based approach is by at least one order of magnitude shorter than the run-time of the BDD-based approach with the best logic area results. In the case if the FBDD-based approach, the area results can be further improved by enabling the DC-based node reduction.

Table 4 presents a comparison between SIS [22] and the FBDD-based approach with respect to the implementation of incompletely specified functions with large DC-sets. Due to the scaling problems of SIS, only the smallest functions that correspond to single outputs of the functions presented in Table 1 could be implemented. The 2nd column reports the number of inputs of each single-output function. The 3rd column (||ON-set|| + ||OFF-set||) gives the sum of the cardinalities of the ON-set and the OFF-set corresponding to each function. The 4th and the 5th columns show the non-terminal node count of the OBDD-based representation of each function. The next 3 columns (SIS) report the resulting gate count, area and the required run-time when the target functions have been implemented directly with SIS. In the last 3 columns (FBDD+SIS), the same parameters are reported for the case

Table 3: Synthesis results obtained using the FBDD-based and the OBDD-based approaches.

Multi-output function	Restrict + Variable Reordering			FBDD		
	Cell area	Synthesis time	Optimization + Synthesis time	Cell area	Synthesis time	Optimization + Synthesis time
p19K	147,074	46m:32s	46m:52s	34,464	1m:56s	3m:27s
	101,332	25m:30s	30m:22s	33,286	1m:29s	18m:57s
	89,681	17m:12s	57m:34s	32,917	1m:30s	23m:37s
p59K	23,075	1m:54s	1m:56s	7,014	30s	41s
	15,198	1m:02s	3m:18s	7,046	37s	1m:04s
	15,292	1m:07s	19m:34s	6,869	29s	1m:39s
p127K	1,349,051	15h:02m	15h:26m	521,814	4h:40m	5h:15m
	1,036,493	7h:06m	18h:22m	507,949	2h:51m	17h:51m
	-	-	-	508,840	3h:07m	19h:41m

Table 4: Comparison between SIS and the FBDD-based approach plus SIS.

Single-output function	#inputs	||ON-set|| + ||OFF-set||	ON-BDD size [#nodes]	OFF-BDD size [#nodes]	SIS			FBDD + SIS		
					#gates	Cell area	Run time	#gates	Cell area	Run time
p1	82	229	6,516	8,592	354	760	28.60s	21	39	0.11s
p2	82	843	21,934	30,621	180	395	3.83s	31	64	0.31s
p3	77	1,708	30,745	63,286	674	1,534	1,046.74s	366	754	26.85s
p4	77	3,652	64,744	128,072	1,145	2,586	3,997.47s	366	820	7.58s

where FBDD-like covers have been generated and later synthesized with SIS. In all the cases, SIS has been run with the rugged script. The statement *full_simplify -m nocomp* has been inserted at the beginning of the script. The library *nand-nor.genlib* has been used.

It is obvious that the second approach scales better and improves dramatically the number of gates and area (between 2 and 19 times). This suggests that the proposed approach enables a much better use of the don't cares which in the descriptions of SIS and MIS are referred to as external don't cares [3][22].

As a final remark, we do not recommend the use of the proposed FBDD-based approach for the synthesis of Boolean functions with no or small DC-sets.

5. CONCLUSION

A new BDD-based logic synthesis procedure for irregular and incompletely specified functions with large DC-sets has been presented, which can help to find efficient multi-level implementations. The problem is reduced to the construction of a minimal FBDD by performing DC-based node reduction and mainly by partitioning the definition space of the target function into a reduced number of subspaces, which may be mapped either to 0 or to 1. Heuristics are used to find near-optimal partitions of the definition space into such subspaces and, consequently, to minimize the path and node count of the resulting FBDD-like covers. Furthermore, this approach is also able to use the DC-set to reduce the number of gates appearing in the circuit description of the non-terminal nodes.

Applying this approach to the synthesis of some benchmark bit-flipping functions [10] resulted in covers whose circuit descriptions contained about 70% less logic operators than the implementations obtained with the *restrict* operator and variable reordering [27]. The synthesis of the resulting circuit descriptions with Synopsys Design Compiler revealed that the FBDD-based approach improves the area figures by a factor between 2 and 3, while the run-time consumption is significantly reduced. Moreover, the proposed method scales better and succeeds to get a better advantage of the DC-set than SIS.

A tool that implements the synthesis approach presented here can be downloaded from [28].

6. ACKNOWLEDGMENTS

We would like to thank Fabio Somenzi, Guenter Bartsch and Christoph Scholl for their helpful comments.

This research work was supported by the German Federal Ministry of Education.

7. REFERENCES

[1] S.B. Akers "Binary Decision Diagrams," *Trans. on Computers*, Vol. C-27, No. 6, 1978, 509-516.

[2] B. Becker "Synthesis for Testability: Binary Decision Diagrams," *STACS. LNCS*, Vol. 577, *Springer Verlag*, 1992, 501-512.

[3] R.K. Brayton et al. "MIS: A Multiple-Level Logic Optimization System," *Trans. on CAD*, 1987, 1062-1081.

[4] R.K. Brayton et al. "Logic Minimization Algorithms for VLSI Synthesis," *Kluver Academic Publishers*, 1997.

[5] R.E. Bryant "Graph-Based Algorithms for Boolean Function Manipulation," *Trans. on Computers*, Vol. C-35, No. 8, 1986, 677-691.

[6] R.E. Bryant "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication," *Trans. on Computers*, Vol. 40, No. 2, 1991, 205-213.

[7] S.-C. Chang et al. "Minimizing ROBDD Size of Incompletely Specified Multiple Output Functions," *European Design and Test Conference*, 1994, 620-624.

[8] O. Coudert et al. "Verification of Sequential Machines using Boolean Functional Vectors," *IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989, 111-128.

[9] O. Coudert, J. Madre "Verification of Synchronous Sequential Machines Based on Symbolic Execution," *Automatic Verification Methods for Finite State Systems*, Springer-Verlag, 1990, 365-373.

[10] V. Gherman et al. "Efficient Pattern Mapping for Deterministic Logic BIST," *Int. Test Conference*, 2004, pp. 48-56.

[11] W. Günther, R. Drechsler "Minimization of Free BDDs," *INTEGRATION, The VLSI Journal*, 32 (1-2), Nov. 2002, 41-59.

[12] J. Hlavicka, P. Fiser "BOOM - a Heuristic Boolean Minimizer," *Int. Conference on Computer Aided Design*, 2001, pp. 439-442.

[13] Y. Hong et al. "Sibling-Substitution-Based BDD Minimization using Don't Cares," *Trans. on CAD of Integrated Circuits and Systems*, 2000, pp. 44-55.

[14] B. Koenemann et al. "A SmartBIST Variant with Guaranteed Encoding," *Asian Test Symposium*, 2001, 325-300.

[15] R. Michalski "On the Quasi-Minimal Solution of the General Covering Problem," *Int. Symposium on Information Processing (FCIP 69) (Switching Circuits)*, Vol. A3, 1969, 125-128.

[16] S. Minato "Binary Decision Diagrams and Applications for VLSI Computer-Aided Design," *Kluver Academic Publishers*, 1997.

[17] A.L. Oliveira et al. "Exact Minimization of Binary Decision Diagrams using Implicit Techniques," *Trans. on Computers*, Vol. 47, No. 11, 1998, 1282-1296.

[18] S. Panda et al. "Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams," *Int. Conference on Computer-Aided Design*, San Jose, CA, 1994, 628-631.

[19] J. Rajski et al. „Embedded Deterministic Test for Low Cost Manufacturing Test", *Int. Test Conf.*, 2002, 301-310.

[20] M. Sauerhoff et al. "On the Complexity of Minimizing the OBDD Size for Incompletely Specified Functions," *Trans. on CAD of Integrated Circuits and Systems*, Vol. 15, 1996, 1435-1437.

[21] C. Scholl et al. "BDD Minimization using Symmetries," *Trans. on CAD of Integrated Circuits and Systems*, Vol. 18, No 2, 1999, 81-100.

[22] E. Sentovich et al. "Sequential Circuit Design using Synthesis and Optimization," ICCD, 1992, 328-333.

[23] T. Shiple et al. "Heuristic Minimization of BDDs using Don't Cares," *Design Automation Conference*, 1994, 225–231.

[24] D. Sieling, I. Wegener "Graph Driven BDDs - a New Data Structure for Boolean Functions," *Theoretical Computer Science*, Vol. 141, No.1-2, 1995, 283-310.

[25] Y. Tang et al. "X-Masking during Logic BIST and its Impact on Defect Coverage," *Int. Test Conference*, 2004, 442-451.

[26] N.A. Touba, E.J. McCluskey „Altering a Pseudo-random Bit Sequence for Scan-Based BIST," *Int. Test Conf.*, 1996, 167-175.

[27] http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

[28] http://www.ra.informatik.uni-stuttgart.de/~ghermanv/benchmarks/index.phtml