A survey of Boolean matching techniques for library binding

Luca Benini and Giovanni De Micheli
Stanford University

Abstract

When binding a logic network to a set of cells, a fundamental problem to be solved is recognizing whether a cell can implement a portion of the network. Boolean matching means solving this task using a formalism based on Boolean algebra. In its simplest form, Boolean matching can be posed as a tautology check. We review several approaches to Boolean matching as well as to its generalization to cases involving don't care conditions and its restriction to specific libraries such as those typical of anti-fuse based FPGAs. We present then a general formulation of Boolean matching supporting multiple-output logic cells.

1 Introduction

Cell-library binding (also called *technology mapping*) is the task of transforming a multiple-level logic representation into an interconnection of components that are instances of cells of a given library. By means of library binding, logic designs can be targeted to different technologies and implementation styles, such as standard cells and gate arrays, including field-programmable gate arrays (FPGAs).

Cell libraries contain the set of logic primitives that are available in the desired design style. Hence the binding process must exploit the features of such a library, in the search for the best possible implementation, which optimizes performance, power consumption, area, *et cetera*. Since different objectives may be of interest, binding is often formulated as a constrained optimization problem, which is computationally intractable [12, 17].

Practical approaches to library binding can be classified into two major groups: *heuristic* algorithms [13, 23] and *rule-based* approaches [11, 20]. In both cases, two subproblems must be solved: *matching* and *selection*. Matching means being able to recognize whether a portion of a multiple-level logic circuit can be implemented by a given cell. Selection means choosing appropriate cells that optimize the figure of merit of interest.

We consider here the matching problem only. Heuristic algorithms for network covering based on tree, graph and Boolean matching as well as rule-based systems have been described elsewhere [12]. In addition, we restrict our attention to libraries of combinational gates.

because register binding is often handled by special methods, e.g., [24]. At first, we consider single-output logic cells, but we shall remove this assumption later.

Early approaches to library binding used graph-based representations of library cells expressing multi-level decompositions into simple Boolean functions, such as two-input NANDs. Matching was implemented as a (sub-) graph isomorphism problem, which can be solved very efficiently when the decomposition graph is a tree. Unfortunately, these approaches suffer from several drawbacks, the most important of which is that these representation are not canonical and thus potential matches may not be detected. Pattern matching approaches to binding have similar drawbacks, when the clusters are multiple-level sum-of-product expressions.

Later approaches to library binding used Boolean matching techniques, which are so called because they are based upon (canonical) Boolean representations of the logic functions. The kernel of Boolean matching techniques is solving a tautology problem, which is co-NP complete. Nevertheless, since in our case the cardinality of the support set of the Boolean functions is small (i.e., most cells have at most 5 or 6 inputs) tautology is solved with little computational burden. In addition, binary-decision diagrams (BDDs) can support extremely efficient tautology checks.

It is the purpose of this paper to review and contrast different methods for Boolean matching for generic and specific libraries. We present also a new approach to Boolean matching that can handle multiple-output logic cells.

2 Background

We assume that the reader is familiar with the basic concepts of Boolean algebra (see [6, 12] for a review) and BDDs [2]. We concentrate here on some specific concepts needed for the understanding of the following material. We denote vectors and matrices in bold, i.e., $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$. A vector whose entries are 1 is denoted by $\mathbf{1}$. We use the symbol $\forall_x$ and $\exists_x$ to designate respectively the *consensus* and the *smoothing* operators. Remember that the consensus operation corresponds to universal quantification and it is computed as $\forall_x f = f_x \cdot f'_x$, while smoothing corresponds to existential quantification and it is computed as $\exists_x f = f_x + f'_x$. Consensus (smoothing) with respect to an array of variables can be computed by repeated application of single-variable consensus (smoothing).

2.1 *Don't care* conditions

The *input controllability don't care set* (CDC) for a Boolean function $f(\mathbf{x})$ (with support variables $\mathbf{x}$) includes all input conditions that are never produced by the environment. We can define a CDC function $f_{CDC} : X \rightarrow \{0,1\}$ whose ON-set is the CDC-set of f.

The *output observability don't care set* (ODC) for f denote all input patterns that represents situations when f is not observed by the environment. We define an ODC function $f_{ODC} : X \rightarrow \{0,1\}$ whose ON-set is the ODC-set of f. The DC function $f_{DC} = f_{ODC} + f_{CDC}$

can be used to express all degrees of freedom available for the implementation $\tilde{f}$ of a single output function, namely:

$$f \cdot f'_{DC} \leq \tilde{f} \leq f + f_{DC}$$

Boolean relations

When considering the minimization of multi-output Boolean functions, the degrees of freedom provided by the environment can be expressed by a *Boolean relation* [33]. Intuitively, Boolean relations are generalizations of Boolean functions, where each input pattern may correspond to more than one output pattern. If we call X the input space and Y the output space, a Boolean relation $\Re$ can be represented by its *characteristic equation*: $\mathcal{X} : X \times Y \rightarrow \{1,0\}$ such that $\mathcal{X}(\mathbf{x},\mathbf{y}) = 1$ if and only if $\mathbf{y}$ is one of the possible outputs of $\Re$ for the input $\mathbf{x}$. We clarify these definitions with an example.

Example 1 *Consider the Boolean relation represented by the following table:*

$in_1 in_2$	$out_1 out_2$

Its characteristic equation is: $\mathcal{X} \quad in'_1 out'_1 out'_2 + in'_1 out_1 out_2 + in_1 out_1 out'_2 = 1$

Libraries

We shall consider in the sequel Boolean functions that model a portion (or cluster) of the circuit, and that are called *cluster functions*. We denote by f a generic cluster function. We call *pattern function* a combinational function modeling a library cell, and we use g to represent a generic pattern function. We assume for now that both cluster and pattern functions are scalar (i.e., have a single output.) This restriction is removed in Section 7.

We denote the library (i.e., the set of pattern functions) by L. We say that an input to a library cell is *stuck-at* 0, if it is connected to ground. This is modeled by replacing by 0 the corresponding variable in the pattern function. We define the *stuck-at* 1 condition in a similar way, *mutatis mutandis*. We say that two (or more) cell inputs are *bridged* together, when they are connected to the same input line. Finally, we say that a library is *closed* under stuck-at and bridging (or closed in brief), if for any stuck-at and/or bridging condition, the corresponding pattern function is equivalent to either the pattern function of another cell in L or to a Boolean constant value (i.e., TRUE or FALSE). Most cell libraries are closed.

Example 2 *A library comprising an inverter, as well as a 4-input, a 3-input and a 2-input NAND cell is closed under stuck-at and bridging conditions. Indeed by shorting two or more cell inputs, or by sticking one (or more) cell inputs to ground or to the power supply line. we have a cell behavior equivalent to another cell in the library or to a constant value. If we remove the 3-input NAND gate from the library, then the library is no longer closed.*

3 Boolean matching

Let us consider a cluster function $f(\mathbf{x})$, with n input variables which are entries of vector $\mathbf{x}$. Let us consider also a pattern function $g(\mathbf{y})$, where the variables in $\mathbf{y}$ are the m cell inputs. For the sake of simplicity, we assume that $n = m$. We will remove this assumption in Section 7. Note that when the cell has more inputs than the cardinality of the support of the cluster function, i.e., $m > n$, then a match requires bridging or sticking-at a constant value some inputs. When considering closed libraries (and most libraries are closed), there exist always a more convenient match, i.e., a simpler cell performing this function. Conversely, when the cell has fewer inputs than n, a match is possible only if some variable in $\mathbf{x}$ is redundant. This can be detected while matching the cluster function and considering *don't care* conditions.

Matching involves comparing two functions and finding an assignment of the cluster variables to the patterns variables. For the sake of explanation, we separate the two issues and we describe first matching two functions defined over the same set of variables. The complete Boolean matching problem will be defined in Section 3.3.

3.1 Input permutation

Consider two functions, f and g, defined over the same variable set $\mathbf{x}$. The two functions are *equivalent* if $f(\mathbf{x}) \overline{\oplus} g(\mathbf{x})$ is a tautology. If the functions are expressed by reduced ordered binary decision diagrams (ROBDDs), such a test can be done in constant time [5].

In general, we are interested in exploring the possible permutations of input variables that yield equivalent behavior. Thus we say that f and g are *$\mathcal{P}$-equivalent* if there exists a permutation operator $\mathcal{P}$ such that $f(\mathbf{x})\overline{\oplus}g(\mathcal{P}\,\mathbf{x})$ is a tautology.

The most simplistic approach to detect a match is to perform $n!$ tautology checks. (Note that $n = m$ is usually small and that cells with more than 6 inputs are rare). Mailhot [26] was the first to propose a method for Boolean matching. He detected tautology by comparing ordered BDDs, and he renounced the canonicity of ROBDDs to save the computing time of reducing the OBDDs of the cluster functions. (Historically, his method preceded the development of efficient ROBDD manipulation tools [5].) To expedite $\mathcal{P}$-equivalence checks, he used filters to prune unnecessary tautology checks (See Section 4.2.) The method can be perfected by using ROBDDs. If each library element is associated with a multi-rooted ROBDD representing all variable permutations, then $\mathcal{P}$-equivalency is again a tautology check, that can be performed in constant time [5].

3.2 Input and output polarity assignment

It is often the case that the *polarity* (also called *phase*) of the inputs and outputs of a combinational network can be altered, because I/Os originate and terminate on registers or I/O pads yielding signals and their complements. Thus it is useful to search for matches with arbitrary polarity assignments, when these reduce the cost of the objective function of interest.

The polarity assignment problem can be explained with the help of a formalism used to classify Boolean functions. Consider all scalar Boolean functions over the same support set of n variables. Two functions f and g belong to the same $\mathcal{NPN}$ class, and are said $\mathcal{NPN}$-equivalent if there is a permutation operator $\mathcal{P}$ and complementation operators $\mathcal{N}_i, \mathcal{N}_o$, such that $f(\mathbf{x}) = \mathcal{N}_o g(\mathcal{P}\ \mathcal{N}_i\ \mathbf{x})$ is a tautology [21]. The complementation operators specify the possible negation of some of their arguments. Similarly, two functions f and g are said to be $\mathcal{N}$-equivalent (or *polarity-related* or *phase-related*) if there exist a complementation operator $\mathcal{N}_i$ such that $f(\mathbf{x}) = g(\mathcal{N}_i\ \mathbf{x})$ is a tautology. $\mathcal{PN}$-equivalence is defined in a similar way.

Boolean matching is often defined in terms of $\mathcal{N}$, or $\mathcal{PN}$, or $\mathcal{NPN}$-equivalence. In principle, $\mathcal{N}$, $\mathcal{PN}$, and $\mathcal{NPN}$-equivalence can be reduced to 2^n, $2^n n!$ and $2^{n+1} n!$ tautology checks. In practice, filters can be used to reduce drastically the number of tries, and early approaches to Boolean matching were relying heavily on filtering [26]. Moreover, canonical forms can be used to check for equivalence in constant time.

Variable assignment and Boolean matching

We distinguish now between the cluster variables $\mathbf{x}$ and the pattern variables $\mathbf{y}$. A matching requires an assignment of cluster to pattern variables, representing the connections between the cluster and the cell. We denote a generic assignment by the *characteristic equation* $\mathcal{A}(\mathbf{x},\mathbf{y}) = 1$ of a *variable mapping function* that maps the variables $\mathbf{x}$ into $\mathbf{y}$.

Example 1 *Consider an assignment which maps each entry in* $\mathbf{x}$ *into the corresponding entry of* $\mathbf{y}$. *Then the characteristic equation is* $\mathbf{x} \overline{\oplus}\ \mathbf{y} = \mathbf{1}$. *Equivalently we can express* $\mathcal{A}(\mathbf{x},\mathbf{y})$ *in scalar form as:* $\prod_{i=1}^{n}(x_i \overline{\oplus}\ y_i) = 1$.

With input permutation, the characteristic equation can be expressed as: $\mathcal{A}(\mathbf{x},\mathbf{y}) = \mathbf{y} \overline{\oplus}\ \mathbf{P}\mathbf{x} = \mathbf{1}$, *where* $\mathbf{P}$ *is a permutation matrix.*

With input permutation and complementation, then $\mathbf{y} \overline{\oplus}\ \mathbf{P}\mathbf{N} \oplus \mathbf{x} = \mathbf{1}$, *where* $\mathbf{N}$ *is a diagonal Boolean matrix.*

The pattern function g under the variable assignment represented by $\mathcal{A}$ is [31]:

$$g_{\mathcal{A}}(\mathbf{x}) = \exists_{\mathbf{y}} \mathcal{A}(\mathbf{x},\mathbf{y}) g(\mathbf{y}) \qquad 1$$

Example 2 *Consider a two-dimensional input space, where:* $\mathbf{x} = [x_1, x_2]^T$ *and* $\mathbf{y} = [y_1, y_2]^T$. *The* $\mathcal{NPN}$ *transformation that maps* x_1 *to* y_2' *and* x_2 *to* y_1 *has the following characteristic equation* $A(x_1, x_2, y_1, y_2) = (x_1 \oplus y_2)(x_2 \overline{\oplus}\ y_1) = x_1 x_2 y_1 y_2' + x_1 x_2' y_1' y_2' + x_1' x_2 y_1 y_2 + x_1' x_2' y_1' y_2 = 1$.

Consider pattern function $g = y_1 y_2$ with the previous assignment. The pattern function under the variable assignment is $\exists_{y_1,y_2} \mathcal{A} g = \exists_{y_1,y_2}(x_1 \oplus y_2)(x_2 \overline{\oplus} y_1) y_1 y_2 = x_2 x_1'$

As a result, a condition for matching is that $f(\mathbf{x}) \overline{\oplus} g_{\mathcal{A}}(\mathbf{x})$is a tautology, or equivalently: $f(\mathbf{x}) \overline{\oplus} \exists_{\mathbf{y}} \mathcal{A}(\mathbf{x},\mathbf{y}) g(\mathbf{y}) = 1$ for any value of x. Therefore there is a Boolean matching if and only if the following formula evaluates to true.

$$\forall_{\mathbf{x}}(f(\mathbf{x}) \overline{\oplus} \exists_{\mathbf{y}}(\mathcal{A}(\mathbf{x},\mathbf{y}) g(\mathbf{y}))) \tag{2}$$

4 Boolean matching algorithms

As outlined in the previous section, finding the correct input permutation and polarity assignment that matches a cluster function with a pattern function may require a large number of tautology tests. Numerous approaches have been proposed to eliminate or reduce the need for iterative tautology check.

4.1 Canonical forms

Burch and Long introduced a canonical form for representing functions modulo input-polarity assignments [4]. This allows us to check for $\mathcal{N}$-equivalence in constant time. The method can be easily extended to cope with $\mathcal{PN}$-equivalence (and $\mathcal{NPN}$-equivalence).

The canonical form for $\mathcal{N}$-equivalence relies on a ROBDD representation and can be seen as an operator (i.e., a Boolean function) whose argument is a Boolean function. Burch and Long named it $C_{\mathcal{N}}$and defined it as follows. For all scalar Boolean functions f and g, then f is $\mathcal{N}$-equivalent to $C_{\mathcal{N}}(f)$. Moreover, if f is $\mathcal{N}$-equivalent to g, then $C_{\mathcal{N}}(f) = C_{\mathcal{N}}(g)$.

Given a function f, its canonical form $C_{\mathcal{N}}(f)$ can be constructed in polynomial time by performing a recursive expansion about its support variables. The structure of the algorithm for forming $C_{\mathcal{N}}$is similar to the ITE algorithm [5, 12]. A description is reported in [4].

Let us consider now matching using the $C_{\mathcal{N}}$operator. The Boolean functions representing a library L can be put in the canonical form $C_{\mathcal{N}}$as a preprocessing step, done once for all for each library. These canonical forms can be stored in a hash table. For each cluster function f of interest, its canonical form $C_{\mathcal{N}}(f)$ must then be computed and checked against the library hash table. This check can be done in constant time.

Unfortunately, no polynomial-time reduction to permutation-canonical form has been proposed so far. In [4] Long and Burch proposed *semi-canonical forms*, that can be computed efficiently but are not unique. For each pattern cell in the library, the (small) set of all its semi-canonical forms is generated and stored once for all in a hash table. The cluster function is matched first constructing one of its semi-canonical forms, then checking for its presence in the library's hash table.

Extensions to cope with $\mathcal{PN}$-equivalence are straightforward, by having the library hash table store the permutation semi-canonical forms in polarity canonical form. Finally, check-

ing for $\mathcal{NPN}$-equivalence is usually done by checking also for $\mathcal{PN}$-equivalence of the complement of f.

4.2 Boolean signatures

A *signature* of a Boolean function is a compact representation that characterizes some of the properties of the function itself. Each Boolean function has a unique signature. On the other hand, a signature may be related to two or more functions. This problem, called *aliasing*, distinguishes signatures from canonical forms.

A necessary condition for a Boolean match is that the corresponding signatures are equal. When signatures are compact, comparing them is an efficient method to determine when two functions do not match, and therefore to reduce the search space for a match. Because of aliasing errors, signatures do not represent sufficient conditions to infer matching. Thus, they are inherently less powerful than canonical forms. Signatures have been used before the introduction of canonical forms, and subsequently in the cases where canonical forms are expensive to compute or their size is too large [27].

Signatures can be based on some properties of the representation of a Boolean function, such as symmetries, unateness, size of co-factors, etc. Some signatures are based on Boolean spectra and they are reviewed in Section 4.3.

Mailhot [26] used signatures to reduce the number of tautology checks needed to determine both $\mathcal{P}$-equivalence and $\mathcal{NPN}$-equivalence. The signatures that he introduced are based on the following facts:

- Any input permutation must associate a unate (binate) variable in the cluster function with a unate (binate) variable in the pattern function.
- Variables or groups of variables that are interchangeable in the cluster function must be interchangeable in the pattern function.

The first condition implies that the cluster and pattern functions must have the same number of unate and binate variables to have a match. Thus integer b is a signature of the function. Moreover, with b binate variables, at most $b! \cdot (n-b)!$ variable permutations need to be considered in the search for a match in the worst case.

Example 3 *Consider the following pattern function from a commercial library:* $g = s_1 s_2 a + s_1 s_2' b + s_1' s_3 c + s_1' s_3' d$ *with* $n = 7$ *variables. Function* g *has 4 unate variables and 3 binate variables.*

Consider a cluster function f *with* $n = 7$ *variables. First, a necessary condition for* f *to match* g *is to have also 4 unate variables and 3 binate variables. If this is the case, only* $3!\ 4! = 144$ *variable orders and corresponding OBDDs need to be considered in the worst case. (A match can be detected before all 144 variable orders are considered). This number must be compared to the overall number of permutations:* $7! = 5040$, *which is much larger.*

The second condition allows us to exploit symmetry properties to simplify the search for a match [26, 28]. Consider the support set of a function $f(\mathrm{x})$. A *symmetry set* is a set of variables that are pairwise interchangeable without affecting the logic functionality. A *symmetry class* is an ensemble of symmetry sets with the same cardinality. We denote a symmetry class by C_i when its elements have cardinality i, $i = 1, 2, \ldots, n$. Obviously classes can be void. The symmetry classes of the pattern functions can be computed beforehand, and they provide a signature for the patterns themselves. Indeed a necessary condition for matching is to have symmetry classes of the same cardinality for each $i = 1, 2, \ldots, n$.

Example 4 *Consider the function* $f = x_1x_2x_3 + x_4x_5 + x_6x_7$. *The support variables of* $f(\mathrm{x})$ *can be partitioned into three symmetry sets:* $\{x_1x_2x_3\}, \{x_4x_5\}, \{x_6x_7\}$. *There are two non-void symmetry classes, namely:* $C_2 = \{\{x_4, x_5\}, \{x_6, x_7\}\}$ *and* $C_3 = \{\{x_1, x_2, x_3\}\}$. *Thus a signature is* $[0, 2, 1, 0, 0, 0, 0]$.

Consider now library cells $g_1 = y_1 + y_2y_3 + y_4y_5 + y_6y_7$ *and* $g_2 = (y_1' + y_2')(y_3 + y_4)(y_5 + y_6 + y_7)$. *The signatures of the cells are respectively* $[1, 3, 0, 0, 0, 0, 0]$ *and* $[0, 2, 1, 0, 0, 0, 0]$. *The signatures of* f *and* g_2 *are equal and indeed* g_2 *is* $\mathcal{NPN}$*-equivalent to* f. *Notice however that in general signature matching is only a necessary condition for Boolean matching.*

Other signatures can be obtained by considering the *satisfy count* of a function, which is the number of its minterms. The satisfy count for f is denoted by $|f|$. The satisfy count can be computed quickly when using ROBDD representations [2]. The satisfy count is an invariant for input permutation and complementation. Thus, it can be used as a signature for determining $\mathcal{P}$-equivalence and $\mathcal{PN}$-equivalence. Note that output complementation changes the satisfy count of a n-input function f from $|f|$ to $2^n - |f|$.

Mohnke and Malik [27] suggested to consider the satisfy counts of the cofactors of a function with respect to its variables for determining $\mathcal{P}$-equivalence and $\mathcal{PN}$-equivalence. Let us consider $\mathcal{P}$-equivalence first. The signature is a vector whose entries are the satisfy counts of the co-factors with respect to the uncomplemented variables. Again, such counts can be computed quickly when using ROBDD representations [2]. Then, a necessary condition for $\mathcal{P}$-equivalence for two functions f and g is that each element of the signature for f has one corresponding and equal element in the signature for g. This can be easily tested by sorting the entries and comparing the sorted signatures. Aliasing occurs when the satisfying count for two or more co-factors are the same. Mohnke and Malik [27] considered *breakup signatures* in these cases, that are based on the distance of minterms from an arbitrary point of the Boolean space. Details are reported in [27].

When considering the $\mathcal{N}$-equivalence problems, the satisfy counts of the co-factors of f with respect to both complemented and uncomplemented variables must be considered. These integer pairs can be arranged in a matrix (with as many rows as the input variables) representing the signature. A necessary condition for $\mathcal{N}$-equivalence of two functions f and g is that each row of the signature for f has the same elements (possibly permuted) as the corresponding row for g. Aliasing occurs when a row has identical elements. To overcome this problem, other signature can be considered that are based on satisfy counts of cofactors with respect to two variables. They are called *component signatures* [27]. Eventually,

when considering the $\mathcal{PN}$-equivalence problems, cofactor signatures can still be used in a straightforward way, but the use of breakup and component signatures is limited.

Similar approaches have been independently proposed by Lai et al. [25], and by Cheng and Marek-Sadowska [8]. In [25] the authors introduced a general method for evaluating the quality of signatures, called *effect/cost ratio*. The *effect* of a signature is the reciprocal of its aliasing probability, while the cost depends on the algorithm used for its computation. (For ROBDD-based algorithms, the cost is usually a low-order polynomial function in the number of nodes). Clearly, signatures with high *effect/cost ratio* should be used. Since exact computation of the *effect* of a signature is sometimes difficult, it can be approximated by the number of different values that the signature may take.

Finally, Tsai and Marek-Sadowska [35] have recently proposed a new set of signatures, which have been proved to be effective when checking for $\mathcal{PN}$-equivalence. Such signatures are based on the *generalized Reed-Muller form* (GRM form) of Boolean functions. GRM forms are useful because they can reveal complex symmetries of input variables and are efficiently constructed with procedures similar to those used for BDDs.

4.3 Spectral methods

There are several spectral representation of Boolean functions [21]. We consider here the Hadamard transform, because it can be efficiently implemented. Consider a n-input Boolean function f. Let $\mathbf{z}$ be a Boolean vector of length 2^n whose i^th entry is $f(bool(i))$, $i = 1, 2, \ldots, 2^n$, being $bool()$ a function returning the binary encoding of an integer. One can view $\mathbf{z}$ as the truth table of f. We then recode the Boolean constants so that they take values $\{1, -1\}$. Namely we define $\mathbf{y} = \mathbf{1} - 2 \cdot \mathbf{z}$.

The spectrum $\mathbf{s}$ of a function f is a vector with 2^n elements, calculated as: $\mathbf{s} = \mathbf{T}^n \cdot \mathbf{y}$, where the Hadamard matrix $\mathbf{T}^k$ of size k is defined recursively as follows:

$$\begin{array}{ll} T^0 & 1 \\ T^k & \begin{bmatrix} T^{k-1} & T^{k-1} \\ T^{k-1} & -T^{k-1} \end{bmatrix} \end{array}$$

Since $\mathbf{T}^n$ is symmetric and has orthogonal columns, its inverse is $1/2^n \cdot \mathbf{T}^n$. Thus a function can be recovered from its spectrum by computing: $\mathbf{y} = 1/2^n \cdot \mathbf{T}^n \cdot \mathbf{s}$ and $\mathbf{z} = 1/2 \cdot (\mathbf{1} - \mathbf{y})$.

Each entry in the spectrum gives some global information about the Boolean function. For example, the first entry is $s_0 = 2^n - 2|f|$ and is called 0^{th}-order coefficient. The following n entries are named first order coefficients and show the correlation of f with its input variables. The remaining coefficients show the correlation of f with the *exclusive or* of some input variables. In particular, j^{th}-order coefficients show the correlation of f with the *exclusive or* of j input variables.

Example 5 *Consider* $f(x_1, x_2, x_3) = x_1x_2 + x_3'$ $(n = 3)$. *Its Hadamard spectrum is:* $[s_0, s_1, s_2, s_{12}, s_3, s_{13}, s_{23}, s_{123}]^T = [-2, 2, 2, -6, -2, -2, -2, 0]^T$. *The* 0^{th} *order coefficient is*

$s_0 = 2^3 - 2 * 5 = -2$. *(In this case* $|f| = 5$*). The first order coefficient* s_1 *is* $s_1 = 5 - 3 = 2$. *Notice that* s_1 *is equal to the number of agreements between* f *and* x_1 *minus the number of disagreements. The second-order coefficient* s_{12} *is* $s_{12} = 3 - 5 = -2$, *i.e. the number of agreements between* f *and* $x_1 \oplus x_2$ *minus the number of disagreements.*

A spectrum uniquely identifies a function. Some operators applied to Boolean functions have specific local effects on the elements of its spectrum vector. In particular, complementing a function corresponds to changing sign to its spectrum. Input complementation correspond to changing the sign of the spectral coefficients related to the complemented variables and input permutation corresponds to permuting spectral entries of the same order. Moreover, substituting the input and/or output of a function with a linear combination (i.e., exclusive or) with other inputs corresponds to swapping spectral elements of different orders. By using these transformations we can group Boolean functions into *disjoint translationally equivalent* classes [14], that are classes closed under these transformations, called here $\mathcal{XNPN}$ because extension of the $\mathcal{NPN}$ concept.

Whereas the $\mathcal{XNPN}$ concept is important for classification of Boolean functions, it is less relevant for matching. Indeed, replacing a cluster with a $\mathcal{XNPN}$-equivalent cell may require the use of additional EXOR cells, thus increasing the cost of a match. If we restrict ourselves to $\mathcal{NPN}$ classes, it can be shown that a $\mathcal{NPN}$ canonical form can be obtained applying a sequence of transformations such that the first $n + 1$ coefficient are made positive and the coefficients from 1 to $n+1$ are in increasing order. Unfortunately, a matching algorithm based on this canonical transformation has one main drawback: since the Hadamard transform has 2^n coefficients, its computational cost is exponential in the number of inputs.

Boolean spectra can be of practical use to matching in two ways. First, they can be used for matching by noticing that two functions are $\mathcal{NPN}$-equivalent if the corresponding spectra are equal modulo complementation and permutation of the coefficients within the same order. Yang [38] proposed a Boolean matching algorithm where permutations and complementations of the elements of a spectrum are attempted, to make it equal to another one. If and only if this process is successful, then the corresponding functions are $\mathcal{NPN}$-equivalent. While the algorithm is generally efficient in early ruling out unfeasible matching, its worst-case performance is exponential.

Second, Boolean spectra can be used as signatures. (Fragments of spectra can also be used: for example the 0^{th}-order coefficient is equivalent to the satisfy count). When considering $\mathcal{P}$, $\mathcal{PN}$, or $\mathcal{NPN}$-equivalent matching, aliasing may arise because a cluster function f may match the spectrum of a pattern function g, being f and g just $\mathcal{XNPN}$ equivalent but not $\mathcal{NPN}$ equivalent. Nevertheless mismatches in Boolean spectra (or in portions thereof) may be used to rule out equivalence of the corresponding Boolean functions. Clarke et al. [9] proposed BDD-based methods for the computation of the spectrum. The main advantage of this approach lies in the high average efficiency of BDD-based manipulation, although the worst case computational complexity is still exponential. Moreover, the authors applied spectral filters to speed-up matching, and gave experimental evidence on the high *effect/cost ratio* of such filters [9].

5 Boolean matching with *don't care* conditions

Multiple-level logic networks have often several *don't care* conditions, that are induced by the interconnection of the network itself. Some of these *don't care* conditions are due to the structuring of the network prior to library binding, while others are due to the binding process itself. When considering *don't care* conditions associated with a cluster function, then multiple matching cells can be found. It is therefore convenient to use *don't care* conditions in the search for the most desirable matching cell.

We consider here both controllability and observability *don't care* conditions associated with the cluster function f and represented jointly as f_{DC}. We refer the reader to [12] for the computation of f_{DC}. We say that a pattern function g matches a cluster function f, if g matches $\tilde{f}$ where $f \cdot f'_{DC} \leq \tilde{f} \leq f + f_{DC}$.

5.1 Compatibility graph

Matching can be defined in terms of $\mathcal{P}$, $\mathcal{NP}$, or $\mathcal{NPN}$-equivalence. The first algorithm for detecting $\mathcal{NPN}$-equivalence using *don't care* conditions was proposed by Mailhot [26]. His approach was limited to functions with four or less support variables ($n \leq 4$). Mailhot made use of a *matching compatibility* graph, which is a directed graph whose vertex set is in one to one correspondence with the $\mathcal{NPN}$-equivalent classes of functions. There are 222 such classes for functions of four variables, but 616126 classes for function of five variables and this explains the limitation to four variables.

Each vertex of the graph is labeled by a representative function of the class. Two vertices are joined by an edge if the corresponding representative functions differ in one minterm. Thus a path between two vertices can be associated with a set of minterms, or equivalently with a Boolean function measuring the difference between the representative functions. We call such function the *error function*.

The vertices are annotated by library elements and their costs, when the pattern functions are in the corresponding $\mathcal{NPN}$class. Given a cluster function f, an $\mathcal{NPN}$-equivalence check can map the cluster function to a vertex $v \in V$. Such vertex always exists, because all $\mathcal{NPN}$classes are represented by the graph. On the other hand, the vertex may correspond or not to a library element. In either cases, matching consists in finding the vertex $u \in V$ associated with the least cost cell that is compatible with the cluster function. The compatibility test reduces to checking whether the error function associated with the path from v to u is included in the *don't care* function f_{DC}, which represents the tolerance on the error. In Mailhot's algorithm, the annotated matching compatibility graph and the paths are computed once for all for any given library and stored. Thus matching with *don't care* conditions requires just an additional inclusion test. Even though most libraries have few cells with more than four inputs, the drawback of this approach is that it does not scale with n due to the size of the graph.

5.2 A formula for Boolean matching with *don't care* conditions

Savoj et. al [31] presented a Boolean condition for matching with *don't care* conditions. Consider a cluster function $f(\mathbf{x})$ and *don't care* set $f_{DC}(\mathbf{x})$ and pattern function $g(\mathbf{y})$. An expression for determining a matching with *don't care* conditions can be derived by extending expression (2) as follows:

$$\forall_{\mathbf{x}}(f_{DC}(\mathbf{x}) + f(\mathbf{x}) \,\overline{\oplus}\, \exists_{\mathbf{y}}(A(\mathbf{x},\mathbf{y})g(\mathbf{y}))) \tag{3}$$

which can be rewritten as:

$$\forall_{\mathbf{x}}(\exists_{\mathbf{y}}(A(\mathbf{x},\mathbf{y})(f_{DC}(\mathbf{x}) + f(\mathbf{x}) \,\overline{\oplus}\, g(\mathbf{y})))) \tag{4}$$

Formula (3) has an immediate meaning: for all the values of the input variables x either the pattern function g with input assignment A must be equal to f or f_{DC} is true. Formula (4) is easily derived from (3).

Example 6 *Consider the cluster function* $f = x_1 \,\overline{\oplus}\, x_2$ *with* $f_{DC} = x_1' x_2$, *and pattern function* $g = y_1 + y_2$. *A variable assignment that assigns* x_1' *to* y_1 *and* x_2 *to* y_2 *yields a match. We verify that with (3). The input assignment function is* $A(\mathbf{x},\mathbf{y}) = (y_1 \oplus x_1)(y_2 \overline{\oplus} x_2)$. *Formula (3) is therefore* $\forall_{\mathbf{X}}(\exists_{\mathbf{y}}((y_1 \oplus x_1)(y_2 \overline{\oplus} x_2)(x_1' x_2 + (x_1 \,\overline{\oplus}\, x_2) \overline{\oplus} (y_1 + y_2))))$. *Computing the smoothing we obtain* $\forall_{\mathbf{X}}(x_1' x_2 + x_1 x_2 + x_1' x_2' + x_1 x_2')$, *that is tautology. thus (3) is satisfied.*

The main problem in using formulae (3) and (4) is to find the variable assignment. Savoj et. al ([31]) proposed an algorithm based upon a search for a variable assignment that satisfies condition (4). To expedite the search, Savoj introduced a class of filters that are valid even for incompletely specified functions. The filters are based on the *satisfy count* of the function and its cofactors. For example, if $|f \cdot f_{DC}'| > |g|$ no matching is obviously possible. The interested reader is referred to [31] for details.

Boolean unification

Boolean unification is the process of finding a solution of a Boolean equation [6]. A method for finding Boolean matching with *don't care* conditions based on Boolean unification was proposed by Chen [7]. A matching is searched for by solving a Boolean *equation* in which the unknowns are the variable matching functions representing input assignments. Note that these functions have been represented implicitly up to now by the characteristic equation $A(\mathbf{x},\mathbf{y}) = 1$. Given $f(\mathbf{x})$, $f_{DC}(\mathbf{x})$ and $g(\mathbf{y})$, we first enforce the matching condition:

$$f(\mathbf{x}) \overline{\oplus} g(\mathbf{y}) + f_{DC}(\mathbf{x}) = 1 \tag{5}$$

which must hold for every x.

The unknowns in this equation are $\mathbf{y} = \phi(\mathbf{x}, \mathbf{r})$, where r is an array of arbitrary functions on x. Solving for the unknowns yields the variable matching. if one exists. The solution

method [7] uses a recursive algorithm reminiscent of the binary branching procedure for Shannon expansion.

If we restrict ourselves to $\mathcal{PN}$ matching, we must limit the generality of the solutions: we allow only functions of the form $\mathbf{y} = \mathbf{PN} \oplus \mathbf{x}$ for some permutation matrix $\mathbf{P}$ and diagonal complementation matrix $\mathbf{N}$. Unfortunately, this constraint is not enforced by equation (5). In order to guarantee that solutions are in the desired form, a branch-and-bound algorithm is proposed in [7] that may degenerate in the worst case to exhaustive enumeration of input permutations and polarity assignments. Although Boolean unification is a general and interesting framework for the description of matching problems, the Boolean unification algorithm presented in [7] does not represent a significant theoretical improvement upon enumerative procedures enhanced by efficient filters.

5.4 Matching using multi-valued functions

One recent and effective approach to Boolean matching with *don't care* [36] exploits *multi-valued functions*. A multi-valued function is a mapping from a n-dimensional space to the Boolean space. The input variables can assume a finite number of values ranging from 1 to n. In symbols, a multi-valued function F is $F : N^n \rightarrow B$, where $N = \{1, 2, ..., n\}$ and $B = \{1, 0\}$. The key idea is to represent admissible input assignments with literals of a multi-valued function, and consequently, sets of admissible input assignments with multi-valued cubes.

Example 7 *The cluster function is $f(x_1, x_2, x_3)$ and the pattern function is $g(y_1, y_2, y_3)$. We consider only input permutations for the sake of simplicity. Assume that admissible input assignments are (x_1, y_2), (x_2, y_1), (x_2, y_2), (x_3, y_1), and (x_3, y_3). This set of admissible input assignments can be represented by the multi-valued cube $x_1^{\{2\}} x_2^{\{1,2\}} x_3^{\{1,3\}}$.*

The cubes of the multi-valued function representing possible input assignments are generated iteratively starting from a sum of products representation of the pattern function g, the cluster function f and its *don't care* function f_{DC}. In the following description we consider only input permutations for simplicity. The procedure has three steps.

First, the functions representing the off-set and on-set of f are obtained: $f_{OFF} = f' \cdot f'_{DC}$ and $f_{ON} = f \cdot f'_{DC}$ and cast in *sum of product* form. The pattern functions are complemented, and stored also in *sum of product* form. We consider matching with one cell represented by g and g'.

Second, for each cube p of f_{ON} and for each cube q of g', a multi-valued function $MvCube(p, q)$ is obtained. $MvCube(p, q)$ expresses the constraint that the only acceptable variable assignments are those that make the two cubes disjoint. This is true if at least one of the variables appearing in p with one polarity is associated with one of the variables appearing in q with opposite polarity. The same procedure is repeated for each cube of f_{OFF} and each cube of g. The intersection of all expressions $MvCube(p, q)$ so generated represents implicitly the set of all possible input assignments that yield a match.

As a last step, feasible input assignments are extracted from the multi-valued representation, by solving a matching problem on a bipartite graph. For details, refer to [36].

Example 8 *Assume that a cube in f_{ON} is $p = x_1 x_2'$ and a cube in g' is $q = y_1' y_2 y_3$. The multi-valued function extracted by p and q is $MvCube(p,q) = x_1^{\{1\}} + x_2^{\{2,3\}}$. The function expresses the constraint that, in order for the two cubes to be disjoint, x_1 can be associated with y_1, or x_2 can be associated with either y_2 or y_3.*

The computational complexity of the procedure is of the order of the product of the cardinalities of the *sum of products* under consideration. This is usually not a serious limitation, because most functions (that may match usual cells) have a manageable sum of product representation, and very effective tools exist for two-level logic minimization. [3]. Moreover, for most libraries, the sum of cubes representations of the pattern functions are usually very small and seldom larger than ten cubes. Another factor affecting the computational complexity is that the intersection of the functions $MvCube(p,q)$ is a *product of sums* form, which may require an exponential number of products to be computed. In [36] the authors propose a heuristic that orders the selection of cubes trying to keep the size of the intersection as small as possible. Extensions of the algorithm to deal with $\mathcal{NPN}$ matching with *don't cares* are straightforward and do not sensibly change the overall complexity.

6 Boolean matching for FPGAs

Binding for field programmable gate arrays may leverage specific techniques, which depend on the architecture of the programmable modules. Whereas binding of look-up table [34] and array based [37] FPGAs does not require matching as defined in this paper, Boolean matching is important for antifuse-based FPGAs [18, 19, 30]. An antifuse-based FPGA consists of an array of programmable logic modules, each implementing a logic function that can be personalized by shorting inputs either to a voltage rail or together, by programming the anti-fuses. The uncommitted module is modeled by a combinational, single-output *module function.*

The library of anti-fuse based FPGAs is represented by all logic functions that can be implemented by personalizing the logic module. Note that such library is closed by definition. As far as library binding is concerned, two strategies can be used. Deriving the entire library and using the Boolean matching techniques described above, or representing the library implicitly by the module function. The first approach is used when some personalizations are discarded, because of some electrical and physical design considerations. We consider the second approach in this section.

Example 9 *Let us consider the FPGAs marketed by Actel Inc.. (See Figure 1). In the Act1 series, the module implements the function: $m_1 = (s_0 + s_1)(s_2 a + s_2' b) + s_0' s_1' (s_3 c + s_3' d)$, while in the Act2 and Act3 series it implements the function: $m_2 = (s_0 + s_1)(s_2 s_3 a + (s_2 s_3)' b) + s_0' s_1' (s_2 s_3 c + (s_2 s_3)' d)$. In both cases, the module is a function of $n = 8$ inputs.*

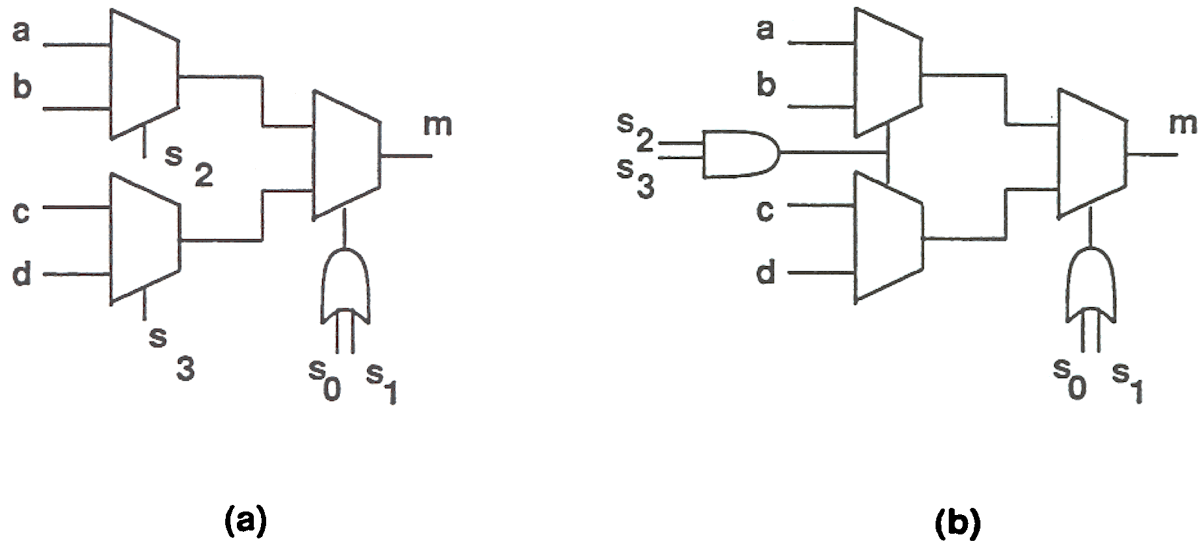

Figure 1: *Act*1 and *Act*2 modules

As an example of programming, by setting $s_0 = s_1 = 1$, function m_1 implements the multiplexer $s_2a + s_2'b$. This is achieved by providing a path from inputs s_0 and s_1 to the power rail through an anti-fuse.

There are about seven hundred functions that can be derived by programming either modules.

For the sake of simplicity, we consider only personalizations by input stuck-ats. Then, the module function can implement any cluster function that matches any of its cofactors. ROBDD representations can be very useful in visualizing and solving this matching problem. Indeed, given an order of the variables of the module function and a corresponding ROBDD representation, its cofactors with respect to the first k variables in the order are represented by subgraphs of the ROBDD. These subgraphs are rooted at those vertices reachable from the root of the module ROBDD along k edges corresponding to the variables with respect to which the cofactors have been taken, or equivalently to those variables that are stuck-at a fixed value by the personalization.

When considering $\mathcal{P}$-equivalence, all variable orders of the module function and the corresponding ROBDDs must be considered to consider all possible personalizations. This can be done by constructing a multi-rooted ROBDD, that encapsulates the library corresponding to the module function. Alternatively, this ROBDD can be represented by a canonical table. Moreover, by $\mathcal{PN}$-equivalence can be efficiently detected by using the canonical forms described in Section 3.2. Extensions to cope with personalization by bridging have also been proposed [15].

Example 10 *Consider the module function $m = s_1(s_2a + s_2'b) + s_1'(s_3c + s_3'd)$ and cluster function $f = xy + x'z$, shown in Figures 2 (a) and (d) respectively. Figure 2 (b) shows the ROBDD of m for variable order: $(s_1, s_2, a, b, c, s_3, d)$ and Figure 2 (c) shows the ROBDD*

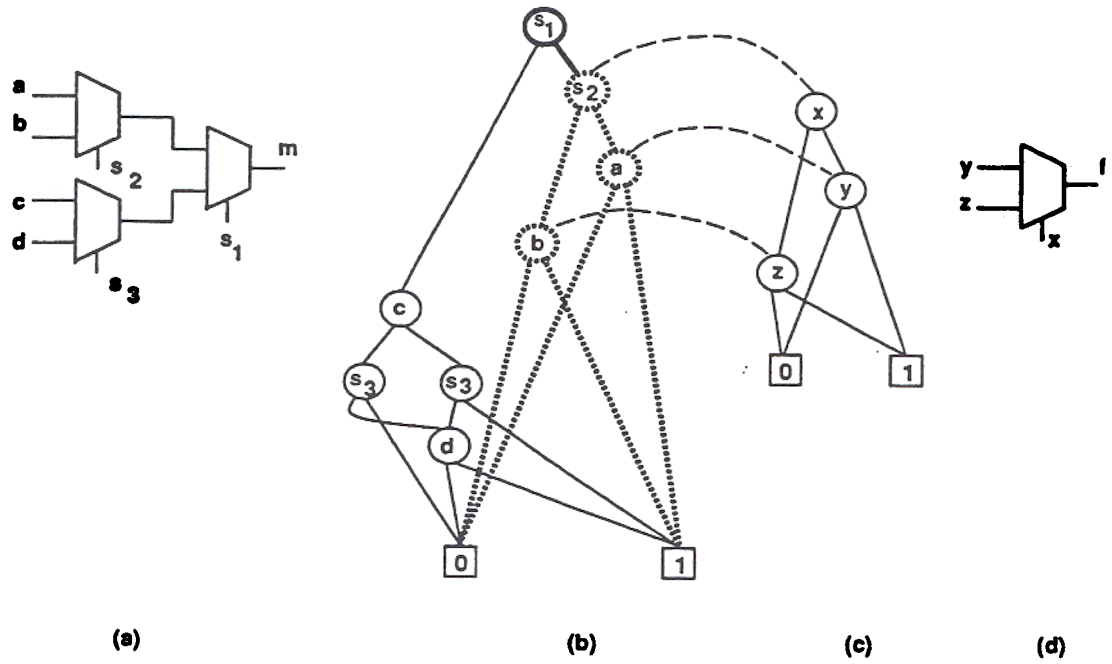

Figure 2: (a) Programmable module. (b) Module ROBDD. (c) Cluster ROBDD. (d) Representation of the cluster function.

of f for variable order: (x, y, z). Since the ROBDD of f is isomorphic to the subgraph of the ROBDD of m rooted in the vertex labeled s_2 (which is the right child of s_1), the module function can implement f by stucking s_1 at 1.

Note that other cluster functions, that can be implemented by the module function, may have ROBDDs that are not isomorphic to any subgraph of the ROBDD of Figure 2 (b). This is due to the fact that a specific variable order has been chosen to construct this ROBDD.

It is important to note that the method just described is applicable to any FPGA library, as long as the module function can be modeled by a single-output logic function and the personalization is performed by sticking-at or bridging cell inputs. We consider next specific methods targeted to the module functions used by some vendors. These methods are faster because module-specific.

Murgai et al. [29] developed Boolean matching procedures specialized to the *Act1* and *Act2* module functions. The algorithms are complex, because both module functions are not quite multiplexing trees. Thus the methods attempt to determine first which inputs of the cluster function should be tied to the multiplexers' selectors (corresponding to the OR gate in Figure 1.) Then they attempt to bind the cluster inputs to the multiplexers' inputs. Two different procedures for *Act1* and *Act2* modules are described in detail in [29].

Fortas et al. [16] addressed the problem of matching the QuickLogic module cell, which can be modeled by a 20-input, 4-output module function. To make the problem more tractable, they considered two fragments of the cell, called M and A respectively and shown in Figure 3. They developed matching algorithms for such fragments. Matching fragment

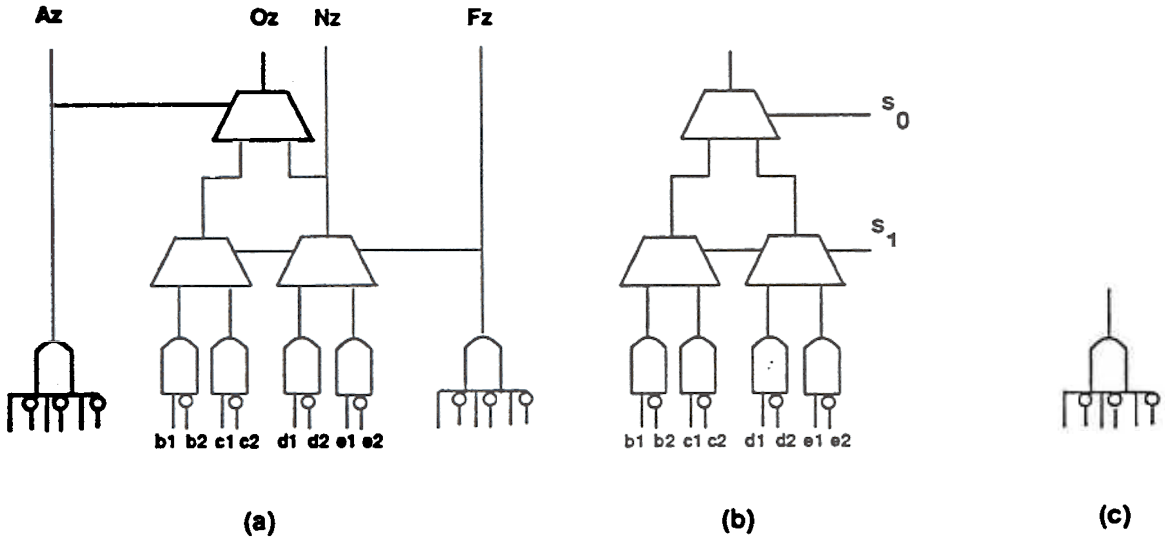

Figure 3: (a) The QuickLogic programmable module. (b) Fragment M. (c) Fragment A.

A is easy, because it implements the conjunction of up to six variables, such that no more than three have the same polarity. Thus this test is always performed first and, if positive, this solution is preferred. When matching fragment M, the cluster variable to bind to the selector input s_0 is chosen first. Then, a test is performed to see if both co-factors can be implemented by gated multiplexers with the same control line. Details of this procedure are reported in [16].

7 A new viewpoint on Boolean matching

As presented in the previous sections, searching for a Boolean match involves some kind of enumeration of the possible variable assignments. The efficacy of some methods is based on clever techniques to reduce the number of alternative solutions that must be tested. The most advanced approaches, namely, those based on canonical forms and multi-valued functions, avoid explicit enumeration by transforming the matching problem into checking the satisfiability of a Boolean formula.

In this section we propose a novel approach that is more general in applicability and retains the desirable characteristic of solving the matching problem by a simple satisfiability check. We remove the restriction on the equality of the cardinality of the support sets for f and g. We consider a pattern function $f(\mathbf{x})$ with n variables and a library cell function $g(\mathbf{y})$ with m inputs. We describe first how we model the variable assignment function using a circuit model. In practice, we use BDD-based symbolic manipulation techniques.

On each input of the cell represented by g we connect the output of a multiplexer whose inputs are the cluster inputs, i.e., the support of f (Figure 4). The control inputs of each multiplexer have the following function: the first $\lceil log_2 n \rceil$ variables control which of the

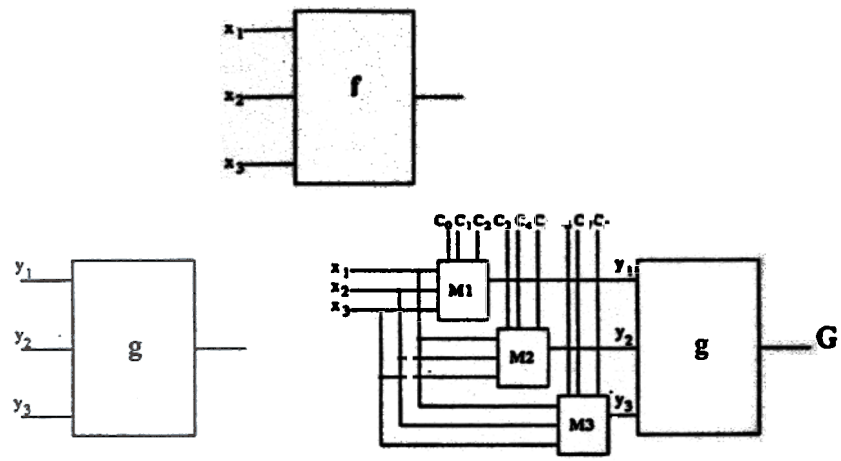

Figure 4: *Transformation of the pattern function $g \rightarrow G$ for matching with cluster function f. The first two control variables of each multiplexer are for permutation control, the last one is for polarity control.*

external n inputs is multiplexed on the input of g. The last control variable controls the polarity of the selected external input.

Example 3 *In the case of Figure 4, consider multiplexer M1. If the control variables C_0 and C_1 are 00, the input x_1 is connected with y_1. If the polarity control variable C_2 is 1, the connection with y_1 will be inverting, therefore x_1' will be seen on y_1.*

From our construction it is clear that the number of control variables needed is $N_c = m(\lceil log_2 n \rceil + 1)$. The key observation is that the control variables $\mathbf{c}$ can be selected in such a way that all $\mathcal{PN}$-equivalent functions of g can be generated. (The inversion of the output can be obtained with one more control variable for the output polarity. We restrict our attention to $\mathcal{PN}$for the sake of simplicity).

In general, the class of functions generated by assignments to $\mathbf{c}$ is actually larger than all input permutations and polarity changes. It includes the cases where two or more of the inputs of g are bridged and connected to the same cluster input with arbitrary polarity or some of the cluster inputs is left unconnected. (Note that this is possible only when that input is redundant because of *don't care* conditions). We call the set of functions that a pattern cell can implement via this multiplexing an *extended-$\mathcal{PN}$($\mathcal{EPN}$)* class. The generalization to $\mathcal{ENPN}$is straightforward.

From an algebraic viewpoint, the introduction of the multiplexers has transformed each pattern function $g(\mathbf{y})$ into a new Boolean function $G(\mathbf{c}, \mathbf{x})$. We define an $\mathcal{EPN}$equivalence relation over the set S of all the Boolean functions with n inputs: $\mathcal{EPN}$-equivalence partitions S into equivalence classes. The set of equivalence classes defined by an equivalence relation is called *quotient set.* Therefore we call $G(\mathbf{c}, \mathbf{x})$ *quotient function* because it implicitly represents an equivalence class (i.e., an element of the quotient set). Indeed all possible assignments of the $\mathbf{c}$ variables individuate all possible functions of $\mathbf{x}$ that belong to the same class as the original library cell function g.

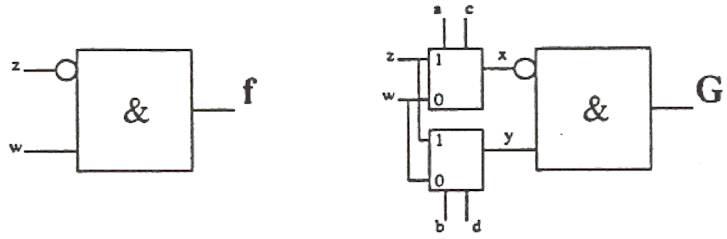

Figure 5: *Pattern function f and quotient function G of Example 3.*

We introduce now a Boolean formula that has at least one satisfying assignment if and only if there exists a function $\mathcal{EPN}$-equivalent to g that is equivalent to f. Intuitively, the formula can be explained by observing that there is an $\mathcal{EPN}$ matching if and only if there exists an assignment $\mathbf{c_0}$ to the control variables $\mathbf{c}$ of $G(\mathbf{c},\mathbf{x})$ such that $G(\mathbf{c_0},\mathbf{x})$ is equal to $f(\mathbf{x})$ for all possible values of $\mathbf{x}$. In other words, the variable assignment represented implicitly by $\mathcal{A}(\mathbf{x},\mathbf{y})$ can be cast in explicit form using $G(\mathbf{c},\mathbf{x})$ and Equation (1) is equivalent to $g_{\mathcal{A}}(\mathbf{x}) = G(\mathbf{c},\mathbf{x})$. Therefore, Boolean matching is represented by:

$$M(\mathbf{c}) = \forall_{\mathbf{x}} [G(\mathbf{c},\mathbf{x}) \overline{\oplus} f(\mathbf{x})] \tag{6}$$

The application of the universal quantifier produces a function of the control variables $\mathbf{c}$. We will call it *matching function*, $M(\mathbf{c})$. Observe two important facts. First, the formula above can be efficiently computed in a fully symbolic way, using BDDs. Second, our procedure finds *all* possible matchings given $f(\mathbf{x})$ and $g(\mathbf{y})$, not just a particular one.

Example 4 *Let the pattern function be $g = x'y$ and the cluster function be $f = wz'$. Figure 5 models $G(a,b,c,d,w,z) = (c \oplus (za + wa'))'(d \oplus (zb + wb'))$, where a,c and b,d are the control variables. We equate f to G:*

$$f \overline{\oplus} G \quad (wz') \overline{\oplus} ((c \oplus (za + wa'))'(d \oplus (zb + wb')))$$

Then we take the consensus of the resulting expression with respect to w and z (the order does not matter), to get $M(a,b,c,d) = ab'c'd' + a'bcd$. The two minterms of $M(a,b)$ describe the two possible variable assignments. Minterm $ab'c'd'$ corresponds to assigning z to x and w to y without any polarity change. Minterm $a'bcd$ corresponds to assigning z to y and w to x changing both polarities. The correctness and completeness of the solution set represented by M can be verified by inspection.

From an implementation standpoint, the matching algorithm operates as follows. First the quotient functions are computed from the ROBDDs of the pattern functions. Thanks to the binary encoding on the control variables of the multiplexers, the size of $\mathbf{c}$ is $O(m \log_2 n)$. This is an important property, because we want to keep the number of variables in the ROBDD representation of G as small as possible for efficiency reasons. Next, given the ROBDD of f, the ROBDD of $G(\mathbf{c},\mathbf{x}) \overline{\oplus} f(\mathbf{x})$ is constructed. The last step is the computation of the consensus over all variables in $\mathbf{x}$ that yields $M(\mathbf{c})$.

When the cluster function is completely specified, traditional matching procedures enhanced with filter appears to be more efficient than our algorithm, because the tautology check is fast and the number of checks is reduced to one in most practical cases [32]. We will show in the following sections that our approach is applicable to a more general class of Boolean matching problems, where traditional techniques cannot be applied.

As a final remark, note that the application of the matching function for binding antifuse based FPGA libraries is straightforward. Only the programmable module function needs to be represented, being the entire library modeled by the quotient functions. Indeed the formulation already takes bridging into account. Stuck-at constant values can be modeled by adding two additional inputs to the multiplexers, each one corresponding to a Boolean constant value.

7.1 Matching incompletely specified functions

Boolean matching with *don't care* conditions can be represented as a straightforward extension of formula (6). Given a cluster function $f(\mathbf{x})$ with *don't cares* represented by $f_{DC}(\mathbf{x})$, there exists a match if there is a satisfying assignment to the following formula:

$$M(\mathbf{c}) = \forall_{\mathbf{x}} [G(\mathbf{c},\mathbf{x}) \overline{\oplus} f(\mathbf{x}) + f_{DC}(\mathbf{x})] \qquad (7)$$

The result of the consensus is again the matching function $M(\mathbf{c})$ representing all possible assignments of the control variables that satisfy the matching condition. Observing the formula, two points are of interest. First, when $f_{DC} = 0$, Equation (7) degenerates to Equation (6). Second, finding a match with or without *don't care* conditions has the same complexity, the only difference being that a different formula must be universally quantified (the number of Boolean variables is unchanged).

Another interesting point is that our procedure can be applied to pattern functions and library cell functions with different number of inputs. We can find a match even when the minimum cost library element g compatible with f has fewer or more inputs than f.

8 Generalized matching

In the previous sections we have discussed the application of our approach to matching problems for single-output functions, where exact solutions have been proposed. We now consider matching of multi-output functions, a problem for which no exact solution has been proposed so far. We present a formula for detecting exactly when Boolean matching of multi-output functions is possible.

Consider the scenario shown in Figure 6. We have a logic network where we identify a set of logic blocks $q_1, q_2, ..., q_l$ (represented by multi-output Boolean function $\mathbf{q}$). Consider the set of logic blocks that are predecessors of $\mathbf{q}$, which we call $f_1, f_2, ..., f_k$ (represented by a multi-output Boolean function $\mathbf{f}$). We focus on the library binding problem for the components $\mathbf{f}$. In the traditional single-output approach, we would bind the components of

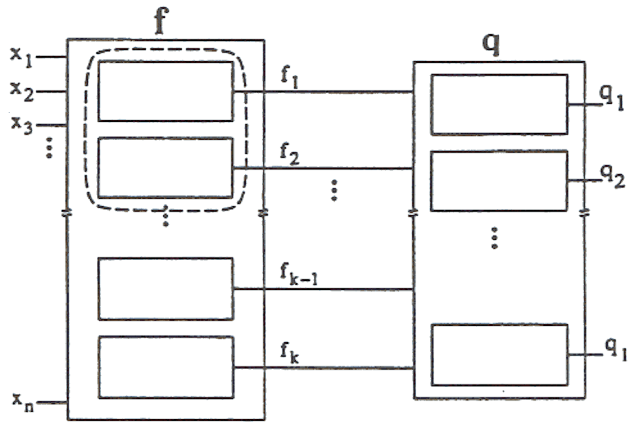

Figure 6: *The general multi-output matching problem. The unbound pattern functions of* f *are enclosed by the dashed line.*

f one by one (possibly considering *don't cares* conditions). Using generalized matching it is possible to perform concurrent binding of two or more cluster functions.

We will show that concurrent binding requires to find a group of single-output library cells (or a single multi-output cell) that satisfy a Boolean constraint expressed as a Boolean relation. This flavor of generalized matching will therefore be called *BR-matching*. Roughly speaking, *BR*-matching is more powerful than matching with *don't cares* , as in the case of the corresponding technology independent optimizations.

In general, the components of f that will be concurrently mapped are called *unbound*. The remaining components of f are considered as *bound* and will be preserved. The two limiting cases of this situation are when only one component is considered unbound and when all components are considered unbound. The first limiting case has already been addressed in the previous section. We will discuss here the second limiting case (f is *fully-unbound*), from which all the intermediate situations can be easily derived. In order to keep the formalism as simple as possible, we will analyze the case of a fully-unbound, two-output cluster function f, as shown in Figure 7. The extension to the general multi-output case is straightforward. Moreover, we will assume that the composite function $h(\mathbf{x}) = q(\mathbf{f}(\mathbf{x}))$ is completely specified and single-output (this hypothesis will be relaxed later).

Whenever $h = 1$, we know that the function q must be $q = 1$ as well (their outputs coincide). The opposite holds when $h = 0$. We can translate this simple observation in a Boolean constraint:

$$q(\mathbf{f}) \overline{\oplus} h(\mathbf{x}) \quad 1 \tag{8}$$

that must hold for each value of x. Notice that the support of q is not x, but the vector f (in our case consisting of f_1 and f_2). We want to test if two library functions g_1 and g_2 (or a two-output library element) can implement block f, without changing the external behavior of h. We will use the quotient functions $G_1(\mathbf{c}_1, \mathbf{x})$ and $G_2(\mathbf{c}_2, \mathbf{x})$ that implicitly represent the $\mathcal{EPN}$ classes. The constraint (8) enforced on all input vectors becomes:

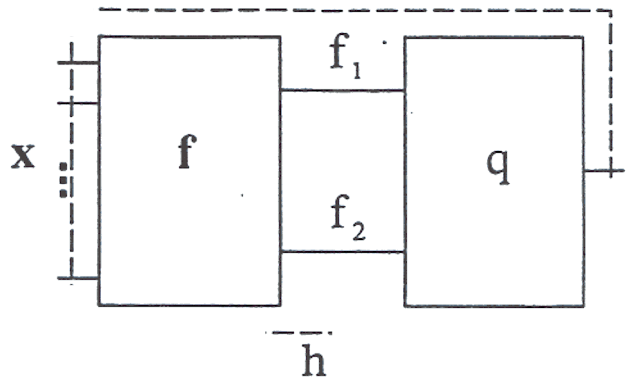

Figure 7: *A typical example of situation where BR-matching is applicable.*

$$M(\mathbf{c_1}, \mathbf{c_2}) = \forall_{\mathbf{x}}[q(G_1(\mathbf{c_1}, \mathbf{x}), G_2(\mathbf{c_2}, \mathbf{x})) \overline{\oplus} h(\mathbf{x})] \tag{9}$$

We clarify the meaning of this Boolean formula through an example.

Example 5 *Consider a block* f *with three inputs* (x_1, x_2, x_3) *and two outputs* f_1 *and* f_2 *that are connected to the inputs of an AND gate. We have* $q = f_1 f_2$. *Assume that the global function is* $h = x_1 x_2' + x_3$. *The Boolean constraint enforced by this structure is:*

$$(f_1 f_2)(x_1 x_2' + x_3) + (f_1' + f_2')(x_1' x_3' + x_2 x_3') \quad \mathbf{1}$$

which is the characteristic equation of the following Boolean relation:

$x_1 x_2 x_3$	$f_1 f_2$
	{10,01,00}
	{11}
	{10,01,00}
	{11}
	{11}
	{11}
	{10,01,00}
	{11}

We consider $\mathcal{EP}$*matching for the sake of simplicity. Our candidate library cell functions for BR-matching of block* f *are* $g_1(y_1, y_2) = (y_1' y_2')'$ *for* f_1 *and* $g_2(y_1, y_2) = (y_1 y_2')'$ *for* f_2. *We need* $N_c = 2 * 4$ *control variables for* $\mathcal{P}$*matching (two control variables for each input of the library cells). The quotient functions are:* $G_1 = ((c_1' c_2' x_1 + c_1' c_2 x_2 + c_1 c_2' x_3)'(c_3' c_4' x_1 + c_3' c_4 x_2 + c_3 c_4' x_3)')'$ *and* $G_2 = ((c_5' c_6' x_1 + c_5' c_6 x_2 + c_5 c_6' x_3)(c_7' c_8' x_1 + c_7' c_8 x_2 + c_7 c_8' x_3)')'$

We replace in the expression of the Boolean constraint all the occurrences of f_1, f_2 *with* G_1 *and* G_2 *and we compute the consensus with respect to* x_1, x_2 *and* x_3. *The resulting matching*

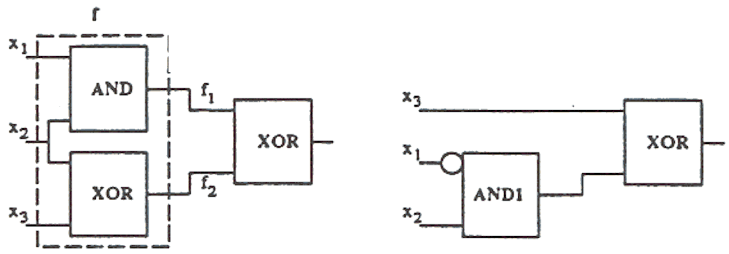

Figure 8: *Application of BR-matching on a multi-output sub-block.*

function M is $M(c_1, c_2, ..., c_8) = c_1'c_2'c_3c_4'c_5'c_6c_7c_8' + c_1c_2'c_3'c_4'c_5'c_6c_7c_8'$ (representing all allowed input assignments). There are two minterms because of the symmetry of the library element g_1.

Notice that formula (9) reduces to formula (6) if the block f has one output. The number of control variables needed is $N_c = m_1(\lceil log_2(n) \rceil + 1) + m_2(\lceil log_2(n) \rceil + 1)$ where m_1 and m_2 are respectively the number of inputs of the library cells g_1 and g_2, and n is the number of inputs.

The general fully-unbound multi-output case (when the block f has k outputs) is the same as the two-output case above described. From a practical standpoint, however, the complexity of BR-matching increases very rapidly with the number of outputs of block f. First, the number of control variables is $O(k\, m\, log_2\, n)$. Second, the number of possible groups of library cells to be tried is $O(|L|^k)$ (where $|L|$ is the number of cells in the library).

Let us now consider the general case in which only some of the cluster functions of block f are unbound. For each unbound component, the quotient function G of a candidate library cell will be inserted in Equation (9), while the bound cluster functions will be left untouched. The advantage of this approach is that it can be applied to situations where the number of control variables needed for fully-unbound BR-matching is too high, or the number of possible groups of library cells is excessive.

Moreover, many libraries include multi-output cells (like full adders and decoders). We can restrict the use of BR-matching to the multi-output cells predefined in our library. In this case, if a matching input variable assignment exists, it must be the same for all output functions, thus the dependence from k of the number of control variables disappears. The quotient function for a multi-output cell does not have more control variables than the quotient function of a single-output cell with the same number of inputs.

Example 6 *We will consider $\mathcal{EP}$-matching for the sake of simplicity. Assume that we have a simple library containing 4 cells: two-input XOR (Cost = 2), two-input AND (Cost = 2), inverter NOT (Cost = 1), two-input AND1 (logic function $in_1 in_2'$, Cost = 3). An implicit cell is the "WIRE" (cost zero). We want to optimize the mapped network of Figure 8. Notice that the mapping cannot be improved with Boolean methods using don't cares because the external don't care set is empty and the XOR on the output does not introduce any ODC on its fan-ins.*

We apply BR-matching to the multi-output cluster function consisting of the first XOR and the AND (enclosed in the dashed box f). The Boolean condition for BR-matching is:

$$\forall_{x_1,x_2,x_3}(h(G_1G_2' + G_1'G_2) + h'(G_1G_2 + G_1'G_2'))$$

Only the cell groups that give a lower cost than the current one (Cost 4) must be considered. The candidate cell groups are listed in the following table.

Cost	Cluster
0	WIRE-WIRE
1	WIRE-NOT NOT-WIRE
2	NOT-NOT WIRE-AND XOR-WIRE WIRE-XOR
3	NOT-AND AND-NOT XOR-NOT NOT-XOR AND1-WIRE WIRE-AND1

5

The number of control variables needed is $4 * 2 = 8$ in the worst case (two two-input cells and three primary inputs), but we will need only $3 * 2 = 6$ for our restricted candidate set of groups.

Applying BR-matching, we find that WIRE-AND1 is a correct replacement. The quotient functions are $G_1 = c_1'c_2'x_1 + c_1'c_2x_2 + c_1c_2'x_3$ (for WIRE) and $G_2 = (c_3'c_4'x_1 + c_3'c_4x_2 + c_3c_4'x_3)(c_5'c_6'x_1 + c_5'c_6x_2 + c_5c_6'x_3)'$ (for AND1). The matching function is $M = c_1c_2'c_3'c_4c_5'c_6'$. The final solution is shown in Figure 8. The optimized network has a lower cost and is fan-out free. Notice that this replacement could not have been found with traditional methods, unless resorting to technology-independent optimizations.

From the example above we can draw some general observations. First, generalized matching is well suited for re-mapping or local optimization. Heuristics must be developed that direct the re-mapping effort on regions of a large network where improvements are required. Second, the efficiency of our procedure will improve if methods that avoid the generation of useless library cell groups are developed.

Finally, we can further generalize *BR*-matching to the case when the block h itself is incompletely specified or described by a Boolean relation [1]. All flavors of generalized matching can be expressed as a satisfiability problem on a suitably extended space.

9 Conclusions

In this work we have reviewed several techniques for Boolean matching to be used by library binding tools. Although traditional techniques based on iterative tautology check enhanced with filters are effective for matching completely specified functions, more advanced approaches have been described that efficiently deal with incompletely specified functions. A *paradigm shift* is taking place: all new techniques solve Boolean matching by transforming it into a satisfiability problem in a different Boolean space.

In the same direction, we described the novel concept of *generalized matching*, that enables concurrent matching of multi-output functions and exploits all local degrees of freedom available for the choice of library cells. Generalized matching is more powerful than Boolean matching with *don't cares* and can be extended to FPGA matching in a straightforward way. We believe that generalized matching is practically appealing as an aggressive optimization step after traditional library binding.

Acknowledgements

This research is supported by NSF under contract number MIP-9421129

References

[1] L. Benini, M. Favalli and G. De Micheli, "Generalized matching, a new approach to concurrent logic optimization and library binding," in *International Workshop on Logic Synthesis*, May 1995.

[2] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, Vol. C-35, No. 8, August 1986, pp. 677-691.

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

[4] J. R. Burch and D. E. Long, "Efficient Boolean function matching," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 408-411, Nov. 1992.

[5] K. Brace, R. Rudell and R. Bryant, "Efficient implementation of a BDD package," in *DAC, Proceedings of the Design Automation Conference*, pp. 40 45, June 1993.

[6] F. Brown. *Boolean reasoning.* Kluwer Academic Publishers, 1990.

[7] K.-C. Chen, "Boolean matching based on Boolean unification," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 346-351, Nov, 1993.

[8] D. I. Cheng and M. Marek-Sadowska, "Verifying equivalence of functions with unknown input correspondence," in *EDAC, Proceedings of the European Design Automation Conference*, pp. 81-85, March 1993.

E. M. Clarke, K. L. McMillan et al., "Spectral transforms for Large Boolean Functions with application to technology mapping," in *DAC, Proceedings of the Design Automation Conference*, pp. 54-60, June 1993.