# A Survey of Boolean Matching Techniques for Library Binding

LUCA BENINI and GIOVANNI DE MICHELI
Stanford University

When binding a logic network to a set of cells, a fundamental problem 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 then present a general formulation of Boolean matching supporting multiple-output logic cells.

Categories and Subject Descriptors: B.6.1 [**Logic Design**]: Design Styles

General Terms: Design, Measurement, Performance

## 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 that optimizes performance, power consumption, and area, etc. Since different objectives may be of interest, binding is often formulated as a constrained optimization problem, which is computationally intractable [De Micheli 1994; Garey and Johnson 1979].

Practical approaches to library binding can be classified into two major groups: *heuristic* algorithms [Detjens et al. 1987; Keutzer 1987] and *rule-based* approaches [Darringer et al. 1981; Gregory et al. 1986]. In both cases, two subproblems must be solved: *matching* and *selection*. Matching

Authors' address: Computer Systems Laboratory, Stanford University, Stanford, CA 94305.

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 interest.

Here we consider 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 [De Micheli 1994]. In addition, we restrict our attention to libraries of combinational gates because register binding is often handled by special methods, e.g., Krishnamoorthy and Mailhot [1994]. At first we consider single-output logic cells, but we will remove this restriction later.

Early approaches to library binding used graph-based representations of library cells expressing multilevel decompositions into simple Boolean functions, such as two-input NANDs [Detjens et al. 1987; Keutzer 1987]. 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 is that these representations are not canonical, and thus potential matches may not be detected. Pattern-matching approaches to binding have similar drawbacks when the cells are modeled by multiple-level sum-of-product expressions.

Later approaches to library binding used Boolean matching techniques, which are so named because they are based upon (canonical) Boolean representations of the logic functions [Mailhot and De Micheli 1993]. The kernel of Boolean matching techniques is solving a tautology problem that is co-NP complete [Garey and Johnson 1979]. 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, tautology is efficiently computed when using binary-decision diagrams (BDDs).

Boolean matching is also applicable to combinational logic verification. Boolean matching (and its extensions) can be used to check the equivalence of two networks, even in absence of information about correspondence of the inputs and the possible change of polarity of some inputs and/or outputs. Nevertheless, verification problems often involve a number of inputs largely superior to those of a typical cell. Therefore the Boolean matching algorithms that are applicable in practice to verification differ from those used in library binding. In this paper we concentrate mainly on the library binding problem.

It is the purpose of this paper to review and contrast different methods for Boolean matching for generic and specific libraries. We also present 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 Brown [1990] and De Micheli [1994] for a review) and BDDs

[Bryant 1986]. We concentrate here on some specific concepts needed to understand 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 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 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 $x$) 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 $f$ denotes all input patterns that represent 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:

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

### 2.2 Boolean Relations

When considering the minimization of multioutput Boolean functions, the degrees of freedom provided by the environment can be expressed by a *Boolean relation* [Somenzi and Brayton 1989]. 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*: $\mathscr{X} : X \times Y \rightarrow \{1,0\}$ such that $\mathscr{X}(\mathbf{x}, \mathbf{y}) = 1$ if and only if $y \in$ Y is one of the possible outputs of $\Re$ for the input $x \in$ X. We clarify these definitions with an example.

*Example 1.* Consider the Boolean relation represented by the table on the following page. The first row of the table means that input 00 corresponds to either output 00 or 11. Note that this condition cannot be expressed by a *don't care* symbol. The second row is similar. The characteristic equation of the Boolean relation is $\mathscr{X} = x'_1 y'_1 y'_2 + x'_1 y_1 y_2 + x_1 y_1 y'_2 = 1$.

| $x_1x_2$ | $y_1y_2$ |
|---|---|
| 00 | {00,11} |
| 01 | {00,11} |
| 10 | {10} |
| 11 | {10} |

### 2.3 Libraries

In the sequel we will consider Boolean functions that model a portion (or cluster) of the circuit and 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 8.

We say that an input to a library cell is *stuck-at* 0 if it is connected to ground. This is modeled by replacing with 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) if for any stuck-at and/or bridging condition the corresponding pattern function is equivalent to either the pattern function of another cell in the library 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 that are entries of vector $x$. Let us consider also a pattern function $g(\mathbf{y})$, where the variables in $y$ are the $m$ cell inputs. For the sake of simplicity, we assume that $n = m$ unless specified otherwise. 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 for some inputs. When considering closed libraries (and most libraries are closed), there always exists 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 $x$ 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 pattern variables. For explanation, we separate the two issues and describe first matching two functions defined over the same set of variables. The complete Boolean matching problem is defined in Section 3.

### 3.1 Input Permutation

Consider two functions, $f$ and $g$, defined over the same variable set $x$. 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 [Brace et al. 1993].

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 and De Micheli [1993] were the first to propose a method for Boolean matching. They detected tautology by comparing ordered BDDs, and renounced the canonicity of ROBDDs to save the computing time of reducing the OBDDs of the cluster functions. (Historically, their method preceded development of efficient ROBDD manipulation tools [Brace 1993].) To expedite $\mathcal{P}$-equivalence checks, they used filters to prune unnecessary tautology checks (see Section 4). The method can be perfected by associating each library element with a multirooted ROBDD representing all variable permutations.

### 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 to be $\mathcal{NPN}$-equivalent if there is a permutation operator $\mathcal{P}$ and complementation operators $\mathcal{N}_i$, $\mathcal{N}_o$, such that $f(\mathbf{x}) \overline{\oplus} \mathcal{N}_o\, g(\mathcal{P}\mathcal{N}_i\mathbf{x})$ is a tautology [Hurst et al. 1985]. 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 N such that $f(\mathbf{x}) \overline{\oplus} g(\mathcal{N}_i\mathbf{x})$ 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 drastically reduce the number of tries, and early approaches to Boolean matching relied heavily on filtering [Mailhot and De Micheli 1993]. Moreover, canonical forms can be used to check for equivalence in constant time.

### 3.3 Variable Assignment and Boolean Matching

In practice, a cluster function is defined over some network variables $x$ and a pattern function is defined over some other variables $y$. A matching requires an assignment of cluster variables 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 $x$ into $y$.

*Example 3.* Consider an assignment that maps each entry in $x$ into the corresponding entry of $y$. 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 $\Pi_{i=1}^{n}(x_i \overline{\oplus} y_i) = 1$. 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 $N$ is a diagonal Boolean matrix.

The pattern function $g$ under the variable assignment represented by $\mathcal{A}$ is [Savoj et al. 1992]:

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

*Example 4.* 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 $\exists_{y_1, y_2} \mathcal{A} g = \exists_{y_1, y_2} (x_1 \oplus y_2)(x_2 \overline{\oplus} y_1) y_1 y_2 = x_2 x'_1$ (Figure 1).

Let us consider $\mathcal{PN}$-equivalence as being the straightforward extension to $\mathcal{NPN}$-equivalence. A condition for matching is that $f(\mathbf{x}) \overline{\oplus} g_{\mathcal{A}}(\mathbf{x})$ is a tautology, or equivalently: $f(\mathbf{x}) \overline{\oplus} \exists_{\mathbf{y}} \mathcal{A}(\mathbf{x}, \mathbf{y}) g(\mathbf{y}) = \mathbf{1}$ for any value of $x$. Therefore, there is a Boolean matching if and only if the following formula evaluates to true.

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

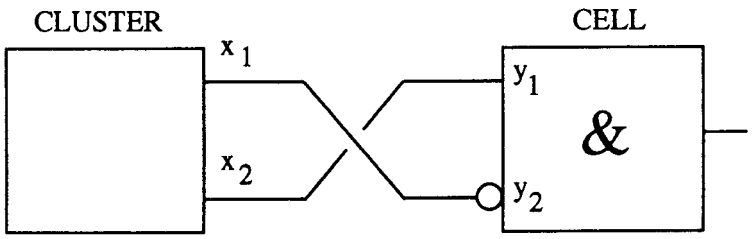


Fig. 1. Input assignment in matching.

## 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 checks.

### 4.1 Canonical Forms

Burch and Long [1992] introduced a canonical form for representing functions modulo input-polarity assignments. This allows us to check for $\mathcal{N}$-equivalence in constant time. This form can be used to check for $\mathcal{PN}$-equivalence (and $\mathcal{NPN}$-equivalence) by testing under all input permutations and output complementation in a straightforward way.

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 $\mathcal{C}_{\mathcal{N}}$ and defined it as follows. For all scalar Boolean functions $f$ and $g$, $f$ is $\mathcal{N}$-equivalent to $\mathcal{C}_{\mathcal{N}}(f)$. 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 $\mathcal{C}_{\mathcal{N}}(f)$ can be constructed in polynomial time by performing a recursive expansion about its support variables. The structure of the algorithm for forming $\mathcal{C}_{\mathcal{N}}$ is similar to the ITE algorithm [Brace et al. 1993; De Micheli 1994]. A description is reported in Burch and Long [1992].

Let us consider matching using the $\mathcal{C}_{\mathcal{N}}$ operator. The Boolean functions representing a library can be put in the canonical form $\mathcal{C}_{\mathcal{N}}$ as a preprocessing step, done once and 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 $\mathcal{C}_{\mathcal{N}}(f)$ must then be computed and checked against the library hash table. This check can be done in constant time.

Canonical forms for representing functions modulo input permutation can be defined in a similar way. For reasons of computational speed, Burch and Long [1992] proposed the use of *semi-canonical forms* for representing permutations. With these forms, which are not unique, $\mathcal{P}$-equivalence can

be tested as follows. For each pattern cell in the library, the (small) set of all its semi-canonical forms is generated and stored once and for all in a hash table. The cluster function is matched by constructing one of its semi-canonical forms and checking for its presence in the library's hash table.

Extensions to cope with $\mathcal{PN}$-equivalence are straightforward: have the library hash table store the permutation's semi-canonical forms in polarity canonical form. Finally, checking for $\mathcal{NPN}$-equivalence is usually done by also checking 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 [Mohnke and Malik 1993].

Signatures can be based on some properties of the representation of a Boolean function, such as symmetries, unateness, size of cofactors, etc. Some signatures are based on Boolean spectra, and are also reviewed in Section 4.

Mailhot and De Micheli [1993] 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:

(1) Any variable assignment must associate a unate (binate) variable in the cluster function with a unate (binate) variable in the pattern function.

(2) 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 (binate) variables to have a match. If we denote by $b$ the number of binate variables, then $b$ is a signature of the function. Obviously, the number of unate variables $(n - b)$ is a signature. Moreover, at most $b! \cdot (n - b)!$ variable permutations need to be considered in the search for a match in the worst case.

*Example 5.* Consider the following pattern function from a 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 also have 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 [Mailhot and De Micheli 1993; Morrison et al. 1989]. Consider the support set of a function $f(\mathbf{x})$. 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 6.* Consider the function $f = x_1x_2x_3 + x_4x_5 + x_6x_7$. The support variables of $f(\mathbf{x})$ 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 [Bryant 1986]. 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 [1993] suggested considering 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 cofactors with respect to the uncomplemented variables. Again, such counts

can be computed quickly when using ROBDD representations [Bryant 1986]. 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 may occur when the satisfying count for two or more cofactors are the same. Mohnke and Malik [1993] considered *breakup signatures* in these cases, which are based on the distance of minterms from an arbitrary point of the Boolean space. Details are reported in Mohnke and Malik [1993].

When considering the $\mathcal{N}$-equivalence problems, the satisfy counts of the cofactors 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 signatures can be considered that are based on satisfy counts of cofactors with respect to two variables. They are called *component signatures* [Mohnke and Malik 1993]. 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.

A similar approach has been independently proposed by Lai et al. [1992], who 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.

Wang et al. [1996] considered the *equivalence signatures* defined over a bipartition $\pi = \{\mathbf{x}_l, \mathbf{x}_r\}$ of the support variables of $f$. The Boolean space spanned by the $\mathbf{x}_l$ variables can be divided into equivalent classes, so that $f(\mathbf{x}_l^a, \mathbf{x}_r) = f(\mathbf{x}_l^b, \mathbf{x}_r)$ for any pair $\{\mathbf{x}_l^a, \mathbf{x}_l^b\}$ in the same class. The number $k$ of such classes is called *communication complexity* of function $f$ w.r.t. $\pi$. Then, given a function $f$ and a variable bipartition $\pi$, the equivalence signature is the set of $k$ pairs, each defined by (i) the satisfy count of an equivalence class and (ii) the cofactor of $f$ w.r.t. the equivalence classes (i.e., the result of partially evaluating $f$ for $\mathbf{x}_l$ corresponding to the class). It was shown by Wang et al. [1996] that equivalence signatures are a generalization of other signatures, and more powerful in screening candidates for matching because different bipartitions $\pi = \{\mathbf{x}_l, \mathbf{x}_r\}$ (with $\mathbf{x}_r$ of increasing size) can be tried. Moreover, equivalence signatures can be computed efficiently from ROBDD representations with variable orders

consistent with the bipartition. This method has applications in verification other than library binding because it can handle functions with more variables (e.g., 10–100) than other methods.

Schlichtmann et al. [1992] proposed the use of different signatures, including *single-fault propagation* signatures. These signatures associate with each variable of a function a triple, counting the patterns that sensitize a fault (that can be propagated to the output on a path with even or odd parity) and those patterns that inhibit the fault sensitization. Cheng et al. [1993] used signatures based on *partner patterns* and *cofactor statistics*, which can be reconduced to single-fault propagation signatures by scaling and modifying the format.

Finally, Tsai et al. [1994] have 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 representations of Boolean functions [Hurst et al. 1985]. We consider here the Hadamard transform because it can be efficiently implemented. Consider an $n$-input Boolean function $f$. Let $z$ be a Boolean vector of length $2^n$ whose $i^{\text{th}}$ 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 $z$ 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 $s$ 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:

$$T^0 \equiv 1$$

$$T^k \equiv \begin{bmatrix} T^{k-1} & T^{k-1} \\ T^{k-1} & -T^{k-1} \end{bmatrix}$$

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 $s$ by computing: $y = 1 / 2^n \cdot \mathbf{T}^n \cdot \mathbf{s}$ 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 7.* 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 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. A second-order coefficient is $s_{12} = 3 - 5 = -2$, representing the number of agreements between $f$ and $x_1 \oplus x_2$ minus the number of disagreements. The third-order coefficient $s_{123} = 0$ measures the number of agreements between $f$ and $x_1 \oplus x_2 \oplus x_3$ minus the number of disagreements.

A spectrum uniquely identifies a function. Unfortunately, using spectra for equivalence checking is not convenient, due to the exponential size of the spectra themselves.

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 a sign to its spectrum. Input complementation corresponds 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) of some inputs corresponds to swapping spectral elements of different orders. By using these transformations we can group Boolean functions into *disjoint translationally equivalent* classes [Edwards 1975] that are classes (of functions) closed under these transformations, called here $\mathcal{XNPN}$, because extension of the $\mathcal{NPN}$ concept.

As a result of the aforementioned properties, $\mathcal{XNPN}$-equivalence can be checked by comparing spectra after the signs have been removed and their elements sorted. Whereas $\mathcal{XNPN}$-equivalence is important for the 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.

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 and De Micheli [1991] proposed a Boolean matching algorithm where complementations and constrained permutations of the elements of a spectrum are attempted to make it equal to another one. Permutations are restricted to be swaps of coefficients of the same order. If and only if this process is successful are the corresponding functions $\mathcal{NPN}$-equivalent. While the algorithm is generally efficient in ruling out unfeasible matching early, 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 the spectrum of a cluster function $f$ may match the spectrum of a pattern function $g$, making $f$ 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. [1993] 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, this group [Clarke et al. 1993] applied spectral filters to speed-up matching, and gave experimental evidence on the high *effect/cost ratio* of such filters.

## 5. BOOLEAN MATCHING WITH *DON'T CARE* CONDITIONS

Multiple-level logic networks often have 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, 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.

Here we consider 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 De Micheli [1994] 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 and De Micheli [1993]. His approach was limited to functions with four or fewer 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 functions of five variables (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

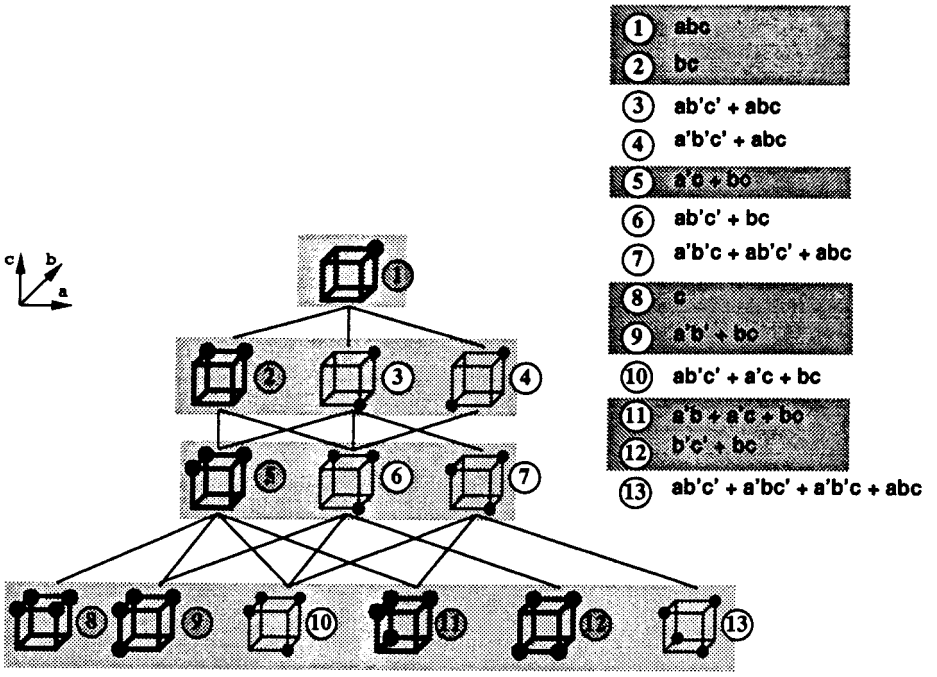


Fig. 2. Matching compatibility graph for 3-variable Boolean space.

function measuring the difference between the representative functions. We call such a function an *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 a 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 case, matching consists of 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 then stored. Thus matching *don't care* conditions only requires 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.

*Example 8.* Consider the matching compatibility graph of Figure 2, where the darker cubes denote vertices corresponding to a hypothetical library. Let the cluster function be $f = xy + xz$ and the *don't care* be

$f_{DC} = x'z'$. The vertex matched to $f$ is $v_5$ and corresponds to a library element. The representative function assigned to $v_5$, i.e., $a'c + bc$, is in the same $\mathcal{NPN}$ class as $f$. (Assign $a'$ to $y$, $b$ to $z$, and $c$ to $x$.) The vertices reachable from $v_5$ are $\{v_9, v_{10}, v_6\}$ because the corresponding paths have minterm sets included in the *don't care* set. Indeed, the errors of using $v_9$, $v_{10}$, and $v_6$, instead of $v_5$, are $a'b'c'$, $ab'c'$, and $b'c'$, respectively, which are all included in $f_{\mathcal{A}DC} = b'c'$ with the variable assignment mentioned above. (Note that the error between $v_5$ and $v_6$ is $b'c'$ because we complement the representative function of $v_6$ and rotate the cube around the $a$ axis.) Only vertex $v_9$ is annotated with a library element. It corresponds to the multiplexer gate because the representative function $a'b' + bc$ is in the same $\mathcal{NPN}$ class as $ab + b'c$.

### 5.2 A Formula for Boolean Matching with *Don't Care* Conditions

Savoj et. al [1992] presented a Boolean condition for matching *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:

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

which can be rewritten as:

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

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

*Example 9.* Consider the cluster function $f = x_1 \overline{\oplus} x_2$ with $f_{DC} = x'_1x_2$, 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 this with Formula (4). The input assignment function is $\mathcal{A}(\mathbf{x}, \mathbf{y}) = (y_1 \oplus x_1)(y_2 \overline{\oplus} x_2)$. Formula (4) is therefore $\forall_{\mathbf{x}}(\exists_{\mathbf{y}}((y_1 \oplus x_1)(y_2 \overline{\oplus} x_2)(x'_1x_2 + (x_1 \overline{\oplus} x_2) \overline{\oplus} (y_1 + y_2))))$. Computing the smoothing, we obtain $\forall_{\mathbf{x}}(x'_1x_2 + x_1x_2 + x'_1x'_2 + x_1x'_2)$, which is a tautology; thus (4) is satisfied (Figure 9).

The main problem in using Formulas (3) and (4) is finding the variable assignment. Savoj et. al [1992] 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 $|f \cdot f'_{DC}| > |g|$, obviously no matching is possible. The interested reader is referred to Savoj et al. [1992] for details.

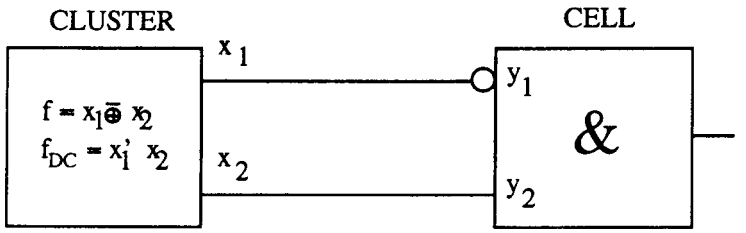


Fig. 3. Input assignment in matching with *don't care* conditions.

### 5.3 Boolean Unification

*Boolean unification* is the process of finding a solution of a Boolean equation [Brown 1990]. A method for finding Boolean matching with *don't care* conditions based on Boolean unification was proposed by Chen [1993]. 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 $\mathcal{A}(\mathbf{x}, \mathbf{y}) = 1$. 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 $\mathbf{x}$.

The unknowns in this equation are $\mathbf{y} = \phi(\mathbf{x}, \mathbf{r})$, where $\mathbf{r}$ is an array of arbitrary functions on $\mathbf{x}$. Solving for the unknowns yields the variable matching, if one exists. The solution method [Chen 1993] uses a recursive algorithm reminiscent of the binary branching procedure for Shannon expansion.

If we restrict ourselves to checking for $\mathcal{PN}$-equivalence, 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 Boolean matrix $N$. Unfortunately, this constraint is not enforced by Equation (5). Similar considerations apply to $\mathcal{P}$-, $\mathcal{N}$-, and $\mathcal{NPN}$-equivalence checking. In order to guarantee that solutions are in the desired form, a branch-and-bound algorithm has been proposed [Chen 1993] 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 [Chen 1993] does not represent a significant improvement upon enumerative procedures enhanced by efficient filters.

### 5.4 Matching Using Multivalued Functions

One recent approach to Boolean matching with *don't care* [Wang and Hwang 1995] exploits *multivalued functions*. A multivalued 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 multivalued function $F$ is $F : N^n \rightarrow B$, where $N = \{1,2, \ldots, n\}$ and $B = \{1,0\}$. The key idea is to represent admissible input assignments with literals of a multivalued function, and consequently, sets of admissible input assignments with multivalued cubes.

*Example 10.* The cluster function is $f(x_1, x_2, x_3)$ and the pattern function is $g(y_1, y_2, y_3)$. For simplicity, we consider only input permutations. 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 multivalued cube $x_1^{\{2\}} x_2^{\{1, 2\}} x_3^{\{1, 3\}}$.

The cubes of the multivalued function representing possible input assignments are generated iteratively, starting from sum of products representation of the pattern function $g$, the cluster function $f$, and its *don't care* function $f_{DC}$. For simplicity, in the following description we consider input permutations only. 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. Then the pattern functions are complemented and stored in *sum of product* form. We consider a cluster function $f$ as 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 multivalued 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 multivalued representation by solving a matching problem on a bipartite graph. For details, refer to Brown [1990].

*Example 11.* 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 multivalued 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 in library binding because most functions (that may match the usual cells) have a manageable sum of

product representation, and very effective tools exist for two-level logic minimization [Brayton 1984]. Moreover, for most libraries, the sum of cube representations of the pattern functions are usually very small and seldom larger than ten cubes. On the other hand, when this method is used for verification, the larger number of inputs can lead to situations where the size of the *sum of product* forms are too large for the method to be practical. 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. Wang and Hwang [1995] proposed 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

Library binding for field programmable gate arrays is often based on some specific techniques, which depend on the architecture of the programmable modules. In particular, binding of look-up tables [Trimberger 1993] and array-based [Wong 1989] FPGA libraries do not require matching as defined in this paper. For example, binding of look-up table FPGAs is often reduced to a decomposition in $k$-input bounded functions, where $k$ depends on the technology, and is usually 4 or 5 [Cong and Ding 1992; Cong and Ding 1996]. On the other hand, Boolean matching is important for antifuse-based FPGAs [Goetting et al. 1995; Green et al. 1993]. 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 *module function*. We concentrate here on FPGAs with single-output modules [Goetting et al. 1995; Green et al. 1993].

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 a 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 12.* Let us consider the FPGAs marketed by Actel Inc. (see Figure 4). In the *Act1* series, the module implements the function: $m_1 = (s_0 + s_1)(s_2a + s'_2b) + s'_0s'_1(s_3c + s'_3d)$, while in the *Act2* and *Act3* series it implements the function: $m_2 = (s_0 + s_1)(s_2s_3a + (s_2s_3)'b) + s'_0s'_1(s_2s_3c + (s_2s_3)'d)$. In both cases, the module is a function of $n = 8$ inputs. As an example of programming, by setting $s_0 = s_1 = 1$, function $m_1$ implements the multiplexer $s_2a + s'_2b$.

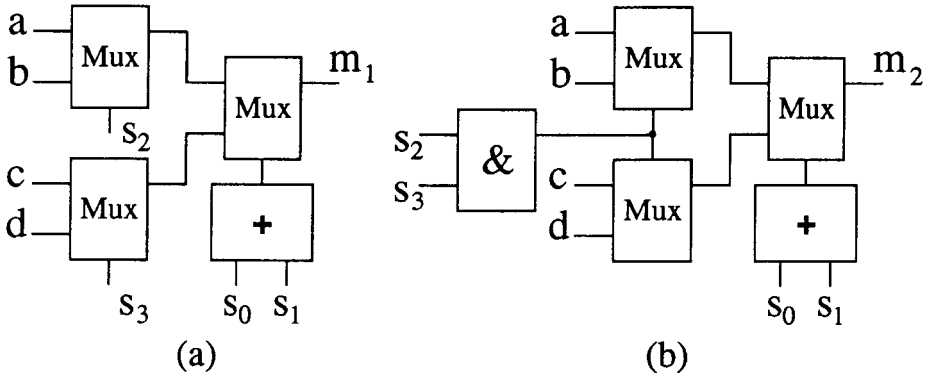


Fig. 4. *Act1* and *Act*2 / 3 modules.

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 700 functions that can be derived by programming either module.

For the sake of simplicity, we consider only personalizations by input stuck-ats, i.e., we exclude bridging. 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 (Figure 13).

When considering $\mathcal{P}$-equivalence, all variable orders of the module function and the corresponding ROBDDs must be taken into account to consider all possible personalizations. This can be done by constructing a multi-rooted ROBDD that captures the library corresponding to the module function. In practice, ROBDDs are stored in canonical tables and the aforementioned operations on graphs can be performed very efficiently by checking table entries [Yang et al. 1997]. Moreover, $\mathcal{PN}$-equivalence can be detected efficiently by using the canonical forms described in Section 3. Extensions to cope with personalization by bridging have also been proposed [Ercolani and De Micheli 1991; Yang et al. 1997]. Another method for matching with stuck-at and bridging connections is mentioned in Section 7.

*Example 13.* Consider the module function $m = s_1(s_2a + s'_2b) + s'_1(s_3c + s'_3d)$ and cluster function $f = xy + x'z$, shown in Figure 13, (a) and (d), respectively. Figure 13(b) shows the ROBDD of $m$ for variable order $(s_1, s_2, a, b, c, s_3, d)$ and Figure 13(c) shows the ROBDD of $f$ for variable order $(x, y, z)$. Since the ROBDD of $f$ is isomorphic to the

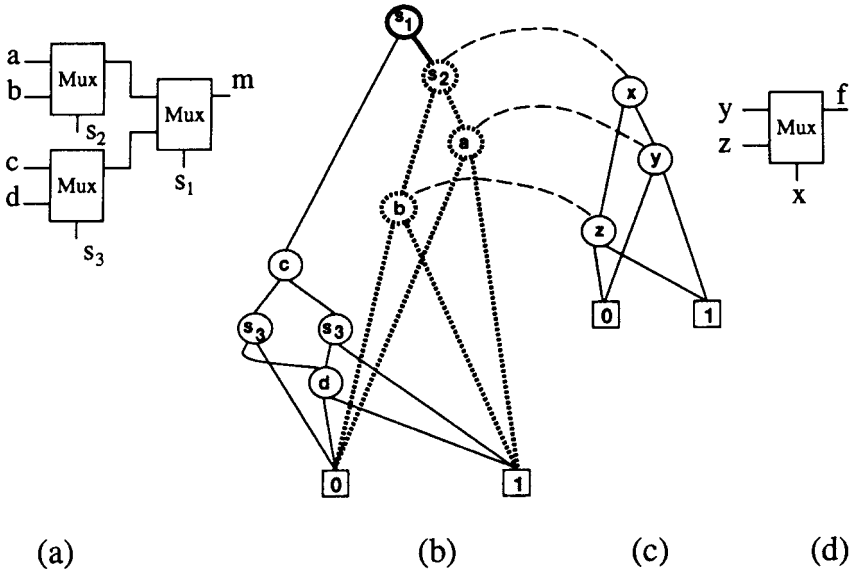


Fig. 5. (a) Programmable module; (b) module ROBDD; (c) cluster ROBDD; (d) representation of the cluster function.

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 sticking $s_1$ 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 13(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. Other approaches to Boolean matching [Fortas et al. 1995; Murgai et al. 1992] are specific to some FPGA modules and exploit the peculiarities of such modules.

## 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 multivalued functions, transform 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 match-

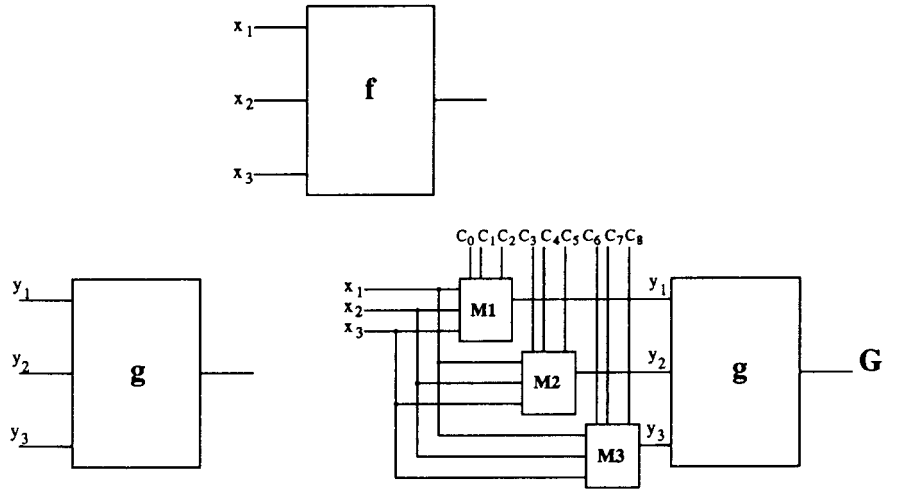


Fig. 6. Transformation of the pattern function $g$ into $G$ for matching with cluster function $f$. The first two control variables of each multiplexer are for permutation control and the last one is for polarity control.

ing problem by a simple satisfiability check. Our matching procedure is completely described in an abstract fashion by Boolean formulas and equations. We describe our approach in a stepwise fashion, starting from the simplest application, namely, matching a completely specified cluster function with a library cell.

We generalize the matching problem in two directions: (i) the cluster function is not required to have the same number of inputs as the pattern function (i.e., $n$ is not necessarily equal to $m$); and (ii) the variable assignment is not required to be a permutation with possible polarity change (e.g., two or more inputs may be bridged together).

A physical interpretation of the matching setup is given by providing each cell input with a polarity control bit (i.e., an exclusive OR gate) and with a multiplexer. The polarity and multiplexer controls are independent for each input and are binary encoded. Namely, the first $\lceil log_2 n \rceil$ variables control which of the external $n$ inputs is multiplexed on the input of $g$. The last control variable controls the polarity of the selected external input. An example is given in Figure 6.

*Example 14.* Consider box M1 in Figure 6, performing controlled complementation and multiplexing. If the control variables are $c_0 = 0$ and $c_1 = 0$, the input $x_1$ is connected with $y_1$. When $c_0 = 0$ and $c_1 = 1$, $x_2$ is connected with $y_1$. When $c_0 = 1$ and $c_1 = 0$, $x_3$ is connected with $y_1$. The last configuration of control variables ($c_0 = 1$, $c_1 = 1$) is unused, and can be assumed to be equivalent to any one of the others. For instance, we assume that when $c_0 = 1$ and $c_1 = 1$, $x_3$ is again connected with $y_1$. The last control variable, $c_2$, defines the polarity of the connection.

If the polarity control variable $c_2$ is 1, the connection with $y_1$ will be inverted, thus either $x'_1$, or $x'_2$, or $x'_3$ 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 variables for 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 larger than the class representative of 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. We call the set of functions a cell can implement with this connection *extended-*$\mathcal{PN}$ ($\mathcal{EPN}$) class. The generalization to $\mathcal{ENPN}$ is straightforward.

From an algebraic viewpoint, the enhanced cell is modeled by 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*. We call $G(\mathbf{c}, \mathbf{x})$ a *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 pattern function $g$.

Boolean matching is easily formulated using the quotient function $G(\mathbf{c}, \mathbf{x})$. We introduce a Boolean formula that has at least one satisfying assignment if and only if the quotient function $G(\mathbf{c}, \mathbf{x})$ (corresponding to the pattern function $g$) is $\mathcal{EPN}$-equivalent to $f$. The formula can be explained intuitivley by observing that there is an $\mathcal{EPN}$ matching if and only if there exists an assignment $\mathbf{c}^*$ to the control variables $\mathbf{c}$ of $G(\mathbf{c}, \mathbf{x})$ such that $G(\mathbf{c}^*, \mathbf{x})$ 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 $G(\mathbf{c}, \mathbf{x})$ can replace $g_{\mathcal{A}}(\mathbf{x})$ in Equation (1). Therefore, the Boolean matching condition 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 variable $\mathbf{c}$. We call it a *matching function,* $M(\mathbf{c})$. Recall that our procedure finds *all* possible matchings given $f(\mathbf{x})$ and $g(\mathbf{y})$, not just a particular one. A minterm of $M$ corresponds to a single $\mathcal{EPN}$ transformation for which $g$ matches $f$. The ON-set of $M$ represents *all* matching $\mathcal{EPN}$ transformations.

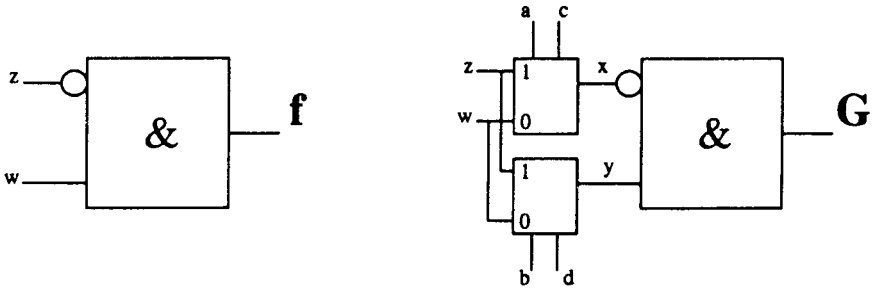


Fig. 7. Pattern function $f$ and quotient function $G$ of Example 3.

*Example 15.* Let the pattern function be $g = x'y$ and the cluster function be $f = wz'$. Figure 7 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 $G$:

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

We then 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 $M(a, b, c, d)$ 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 $x$ while 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 constructed from the pattern functions and stored as ROBDDs. 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 $M(\mathbf{c})$. Observe that, thanks to the binary encoding of the control variables, the size of $\mathbf{c}$ is $O(m \log_2 n)$. This is an important property because for efficiency we want to keep the number of variables in the ROBDD representation of $G$ as small as possible.

When the cluster function is completely specified, traditional matching procedures enhanced with filters appear to be more efficient than using the quotient function since the tautology check is fast and the number of checks is reduced to one (or few) in most practical cases [Schlichtmann et al. 1993]. However, our approach is applicable to much more general Boolean matching problems, where traditional techniques cannot be applied. We now extend the basic matching procedure to progressively more general matching problems.

The first and most straightforward extension is Boolean matching with *don't care* conditions. 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:

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

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 is done by computing a simple Boolean formula, and the computational burden is the same. Moreover, our procedure can be applied to pattern and cluster functions with different numbers of inputs. We can find a match even when the minimum cost library element $g$ compatible with $f$ has fewer or more inputs than $f$. Thus, the application of this matching procedure to binding antifuse based FPGA libraries is straightforward. Only the programmable module function needs to be represented, being the entire library modeled by the corresponding quotient function. 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.

We now describe a further extension of the matching formulation, which allows us to combine matching and cell selection in a single step. This extension is important because the generalized formula denotes all cells and corresponding variable assignments that match a cluster. Given their costs in some metric, the locally-best replacement for the cluster can be chosen in a single step. This contrasts traditional methods requiring an iterative inspection of all (matching) cells.

Combined matching and cell selection is captured by an extended quotient function, representing the entire library, as shown by the following example.

*Example 16.* The extended quotient function is shown pictorially for a simple 3-cell library in Figure 8. In addition to input multiplexing and complementation, the cell outputs are also multiplexed and (possibly) complemented. Multiplexer $Mout$ has three control variables: $c_9$ and $c_{10}$ are used to select which library cell is connected to the outpts, $c_{11}$ selects the polarity of the connection.

The extended quotient function $L(\mathbf{c}, \mathbf{x})$ has $\lceil \log_2 N_{lib} \rceil + 1$ additional control variables for cell selection, where $N_{lib}$ is the number of cells in the library. When $M(\mathbf{c})$ is computed using Equation (6) or (7), one minterm of $M(\mathbf{c})$ not only identifies an input permutation and polarity assignment, but it also specifies which library cell the input assignment matches.

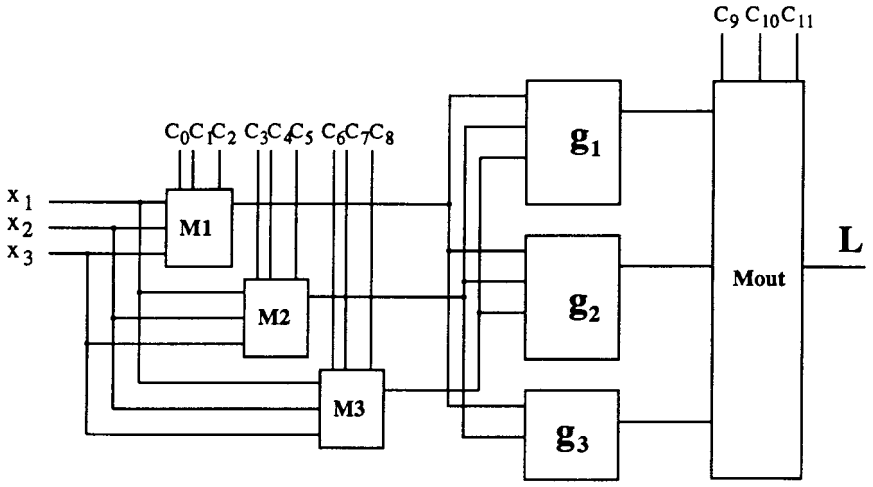


Fig. 8. Quotient function for cell selection and matching.

Since library cells have in general different numbers of inputs, to construct the quotient function for a library, we need as many input-control multiplexers as the maximum number of inputs of any cell in the library $m_{\max}$. Hence, the number of control variables needed for the construction of the quotient function is $\lceil \log_2(N_{lib}) \rceil + 1 + m_{\max} \lceil \log_2(n) \rceil + m_{\max}$.

*Example 17.* Consider a simple library containing three cells $g_1$, $g_2$, and $g_3$. The quotient function for matching and cell selection is shown in Figure 8. The output multiplexer function is represented by block $Mout$ with three control variables, $c_9$, $c_{10}$, and $c_{11}$. If $c_9 = 0$ and $c_{10} = 0$, cell $g_1$ is selected. Cells $g_2$ and $g_3$ are selected with $c_9 = 1$, $c_{10} = 0$, and $c_9 = 1$, $c_{10} = 1$, respectively. Control variable $c_{11}$ selects the polarity of the connection: inverting if $c_{11} = 1$, and noninverting otherwise. In the construction of $L(\mathbf{c}, \mathbf{x})$, we need three input multiplexers because $m_{max} = 3$. Gate $g_3$ has only two inputs, hence it is connected to only two input multiplexers. Consider a configuration of control variables $\mathbf{c}^* = [0,0,0,1,0,0,1,1,0,0,0,0]$. Configuration $\mathbf{c}^*$ corresponds to selecting cell $g_1$ (with no output inversion) with input $x_1$ connected to its first (topmost, in Figure 8) input, $x_2$ is connected to its second input and $x_3$ is connected to its third input. No input is inverted.

## 8. GENERALIZED MATCHING

We now remove the restriction on dealing with single-output clusters and cells. We extend our approach to cope with concurrently matching the multiple outputs of a cluster, and call it *generalized matching*. No exact solution has been proposed so far in the literature for this problem.

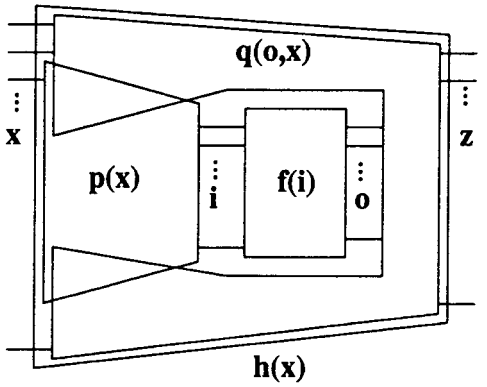


Fig. 9. A multioutput cluster function embedded in its environment.

Generalized matching can achieve two practical goals. First, concurrent matching can yield a binding with a lower cost, as compared to matching each cluster output independently. Second, we can attempt to match multiple-output cells to multiple-output clusters.

We address concurrent matching first. Consider the Boolean network shown in Figure 9. We have a multioutput cluster function $\mathbf{f}(\mathbf{i})$ embedded in a larger Boolean network. If we were to use a traditional matching algorithm, we would match the cluster outputs (i.e., the components of $\mathbf{f}$) one at a time (possibly considering *don't care* conditions). Note that generalized matching is *not* equivalent to a sequence of single-output matching with *don't cares*. There are solutions that can be found only if we concurrently match the multiple-output cluster function to two or more pattern functions. Thus, generalized matching may lead to an overall lower-cost binding.

Generalized matching requires finding a group of single-output pattern functions that satisfy a constraint expressed as a *Boolean relation*. In the following, we adopt a formalism similar to that used by Watanabe et al. [1996] in their work on multioutput Boolean minimization. Indeed, our approach can be seen as an extension of similar ideas to the realm of library binding. We call $\mathbf{x}$ and $\mathbf{z}$ the arrays of Boolean variables at the inputs and the outputs of the network that embeds the cluster function $\mathbf{f}$. The functionality of such a network is represented by the Boolean function $\mathbf{h}(\mathbf{x})$. The inputs of the cluster function can be seen as a function $\mathbf{p}(\mathbf{x})$ of the inputs $\mathbf{x}$. The function $\mathbf{q}(\mathbf{o}, \mathbf{x})$ describes the behavior of the outputs $\mathbf{z}$ when the outputs of the cluster functions are seen as additional primary inputs.

From $\mathbf{h}$, $\mathbf{p}$, and $\mathbf{q}$ we obtain three characteristic functions $H$, $P$, and $Q$, defined as follows:

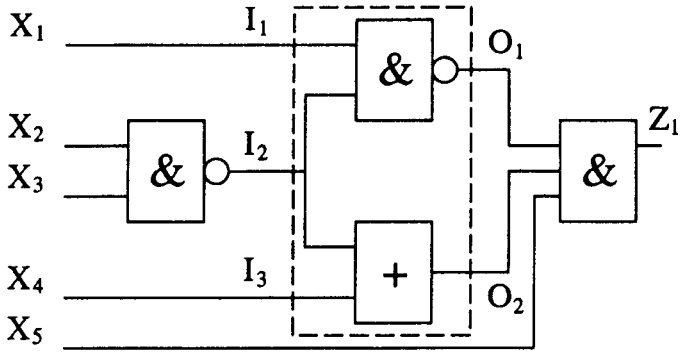


Fig. 10. A two-output cluster function embedded in a Boolean network.

$$H(\mathbf{x}, \mathbf{z}) = \prod_j h_j(\mathbf{x}) \overline{\oplus} z_j \tag{8}$$

$$P(\mathbf{x}, \mathbf{i}) = \prod_j p_j(\mathbf{x}) \overline{\oplus} i_j \tag{9}$$

$$Q(\mathbf{o}, \mathbf{x}, \mathbf{z}) = \prod_j q_j(\mathbf{o}, \mathbf{x}) \overline{\oplus} z_j \tag{10}$$

The characteristic functions fully describe the environment around the multioutput function **f**. In particular, they enable the computation of a Boolean relation representing the complete set of *compatible functions* of **f**, i.e., functions that can implement **f** without changing the input-output behavior of **h**. Watanabe et al. showed that the characteristic function $\mathcal{F}$ of the Boolean relation can be obtained by the following formula [Watanabe et al. 1996]:

$$\mathcal{F}(\mathbf{i}, \mathbf{o}) = \forall_{\mathbf{x}, \mathbf{z}}[(P(\mathbf{x}, \mathbf{i}) \cdot Q(\mathbf{o}, \mathbf{x}, \mathbf{z})) \Rightarrow H(\mathbf{z}, \mathbf{x})]$$

In words, $\mathcal{F}$ represents the set of values of **i** and **o** such that if $Q$ is true and $P$ is true, then $H$ is true for all possible values of **x** and **z**. Formula (1) allows us to find all functions **f** that, when composed with **p** and **q**, produce exactly function **h**. There are generally many functions with this property. These functions are represented by a Boolean relation, and $\mathcal{F}$ is the characteristic function of such a relation.

*Example 18.* Consider the Boolean network shown in Figure 10. The dashed rectangle encloses the multioutput cluster function $\mathbf{f} = [f_1, f_2]^T$, $f_1 = (i_1 i_2)'$, $f_2 = i_2 + i_3$. Function **h** has a single output $h_1 = x_5(x_4 + x'_2 + x'_3)(x'_1 + x_2 x_3)$. Function **q** has three inputs and one output: $q_1 = x_5 o_1 o_2$. Function $\mathbf{p} = [p_1, p_2, p_3]^T$ has four inputs and

| $i_1 i_2 i_3$ | $o_1 o_2$ |
|---|---|
| 000 | {10,01,00} |
| 001 | {11} |
| 010 | {11} |
| 011 | {11} |
| 100 | {10,01,00} |
| 101 | {11} |
| 110 | {10,01,00} |
| 111 | {10,01,00} |

three outputs: $p_1 = x_1$, $p_2 = (x_2 x_3)'$ and $p_3 = x_4$. Applying Equation (1), we obtain the Boolean relation representing all degrees of freedom in the implementation of **f**. For ease of understanding, it is in tabular form (see above). The characteristic function of the Boolean relation is $\mathcal{F}(i_1, i_2, i_3, o_1, o_2) = (o'_1 + o'_2)(i_1 i_2 + i'_2 i'_3) + o_1 o_2 (i'_1 i_2 + i'_2 i_3)$.

Once $\mathcal{F}$ has been computed by Formula (1), we can derive the generalized matching equation. Assume that the multioutput cluster function **f** has $n_o$ outputs. We call $\mathcal{L}_k$ the characteristic functions of $n_o$ quotient functions, one for each output of the multioutput cluster function **f**. Namely, $\mathcal{L}_k(\mathbf{c}_k, \mathbf{i}, o_k) \equiv L(\mathbf{c}_k, \mathbf{i}) \overline{\oplus}\, o_k$, $k = 1,2, \ldots, n_o$. Generalized matching is described by the following formula:

$$M(\mathbf{c}) = \forall_{\mathbf{i}} \exists_{\mathbf{o}} \left( \mathcal{F}(\mathbf{i}, \mathbf{o}) \cdot \prod_{k=1}^{n_o} \mathcal{L}_k(\mathbf{c}_i, \mathbf{i}, \mathbf{o}_k) \right) \tag{12}$$

To understand the formula, observe that the conjunction between $\mathcal{F}$ and all $\mathcal{L}_k$, $k = 1,2, \ldots, n_o$, followed by existential quantification of the output variables, is equivalent to the condition that for any output vector $\mathbf{o}^* = [o_1^*, o_2^*, \ldots, o_{n_o}^*]^T$, the quotient functions associated with each component assume a consistent value: $L_1 \equiv o_1^*$, $L_2 \equiv o_2^*$, $\ldots$, $L_{n_o} \equiv o_{n_o}^*$. The universal quantifier on the inputs **i** enforces the condition for all possible input values.

Notice that the quotient functions $L(\mathbf{c}_k, \mathbf{i})$ have distinct control variables. In other words, the complete vector of control variables **c** on the left-hand side of Equation (1) is the concatenation of the control variables of all $n_o$ quotient functions: $\mathbf{c} = [\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_{n_o}]^T$. The ON-set of $M(\mathbf{c})$ includes all configurations of control variables representing the ways in which the library cells can be connected so as to obtain a final implementation of **f** contained in relation $\mathcal{F}$.

*Example 19.* Consider the two-output, three-inputs cluster function **f** introduced in Example 18, and the three-cell library of Example 17 with the corresponding quotient function $L(\mathbf{c}, \mathbf{i})$. To perform generalized matching, we need to instantiate two quotient functions $L_1(i_1, i_2, i_3, c_0, \ldots,$

$c_{10}, c_{11}$) and $L_2(i_1, i_2, i_3, c_{12}, \ldots, c_{23})$. Notice that $L_1$ and $L_2$ have different supports, but are otherwise identical. The characteristic functions of the quotient functions are $\mathcal{L}_1(i_1, i_2, i_3, o_1, c_0, \ldots, c_{11}) = L_1 \overline{\oplus}\, o_1$ and $\mathcal{L}_2(i_1, i_2, i_3, o_2, c_{12}, \ldots, c_{23}) = L_2 \overline{\oplus}\, o_2$. The generalized matching equation

$$M(c_0, \ldots, c_{21}) = \forall_{i_1, i_2, i_3} \exists_{o_1, o_2} (\mathcal{F}(i_1, i_2, i_3, o_1, o_2) \cdot \mathcal{L}_1(i_1, i_2, i_3, o_1, c_0, \ldots, c_{11}) \cdot \mathcal{L}_2(i_1, i_2, i_3, o_2, c_{12}, \ldots, c_{23}))$$

where $\mathcal{F}(i_1, i_2, i_3, o_1, o_2)$ is the characteristic function of the Boolean relation for **f** computed in Example 18. A minterm $\mathbf{c}^*$ of $M$ uniquely identifies two library cells and an input assignment for all their inputs.

Generalized matching is performed by directly implementing Equation (1) using standard BDD operators. The number of control variables in Equation (1) increases with $n_o$. More precisely, the number of control variables is $N_c = n_o(\lceil (\log_2 N_{lib}) \rceil + 1 + m_{\max} \lceil \log_2(n) \rceil + m_{\max})$, where $N_{lib}$ is the number of cells in the library, $n$ is the number of inputs of **f**, and $m_{\max}$ is the maximum number of inputs of a library cell. The term multiplied by $n_o$ is the number of control variables contributed by each quotient function. The first logarithmic contribution accounts for the control variables for cell selection, the constant "1" is for output polarity assignment, the log-linear contribution is for input permutation, and the linear contribution is for input polarity assignment.

*Example 20.* Referring to the multioutput target function introduced in the previous example, $\mathcal{F}$ has two output ($n_o = 2$) and three inputs ($n = 3$). Assume that the library has 75 cells ($N_{lib} = 75$) and that the cell with the largest support in the library has 5 inputs ($m_{\max} = 5$). The computation of the matching function $M$ for Boolean relation $\mathcal{F}$ requires $N_c = 2\ (\lceil \log_2 75 \rceil + 1 + 5 \lceil \log_2 3 \rceil + 5) = 2\ (7 + 1 + 10 + 5) = 46$ control variables.

From a practical standpoint, the complexity of generalized matching increases rapidly with the number of outputs of **f**. The number of control variables can be drastically reduced if symmetry is considered for input assignments and filters are applied to reduce the number of candidate library cells in the construction of the quotient function. In this overview, we do not focus on implementation details and efficiency issues. The enhanced power of generalized matching will be clarified through an example.

*Example 21.* Assume that we have a simple library with 4 cells: two-input XOR ($Cost = 2$), two-input AND ($Cost = 2$), inverter NOT ($Cost = 1$), and two-input AND1 (logic function $g = in'_1 in_2$, $Cost = 3$). An implicit cell is the "WIRE"(cost zero). We want to optimize the

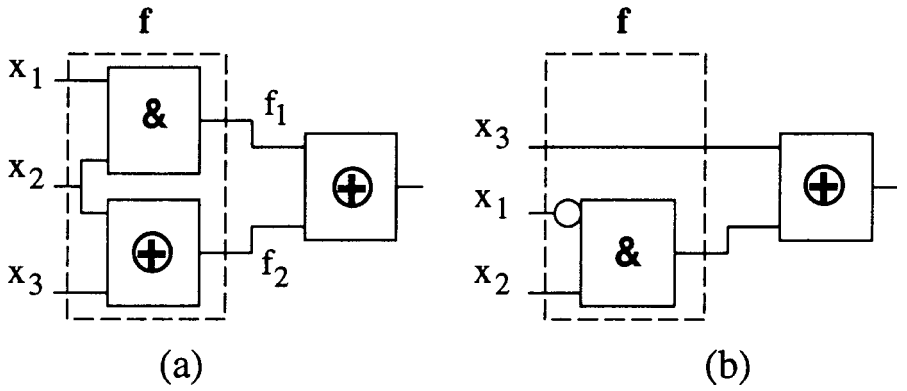


Fig. 11. An example of the effectiveness of generalized matching.

mapped network of Figure 11 (a). Notice that the binding 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 generalized matching to the multioutput cluster function consisting of the first XOR and the AND (enclosed in the dashed box **f**). The number of control variables needed is $N_c = 2(\lceil \log_2 4 \rceil + 1 + 2\lceil \log_2 3 \rceil + 2) = 18$. Applying generalized matching and examining the cost of the solutions (i.e., the ON-set of $M(\mathbf{c})$), we find that WIRE on output 1 and AND1 on output 2 is a correct replacement. The final solution is shown in Figure 11(b). The reader can verify its correctness by inspection. The optimized network has a lower cost and is fan-out-free. Notice that this replacement could not have been found with traditional matching, even with *don't cares,* unless resorting to technology-independent optimizations.

We consider next the application of generalized matching to binding multiple-output cells, which are common in many semicustom libraries (e.g., full adders, decoders). Multiple-output cells implement multiple-output pattern functions over the same set of inputs. As a result, the variable assignment used in matching must be the same for all components of the pattern function. This constraint has a beneficial effect in reducing the number of control variables. Namely: $N_c = n_o(\lceil \log_2 N_{libOut} \rceil + 1) + m_{\max}\lceil \log_2 n \rceil + m_{\max}$. The first term accounts for the $n_o$ output multiplexer functions (with output polarity assignment). $N_{libOut}$ is the total number of outputs of all multioutput library cells. The second and third terms account for the input permutations and polarity assignments.

*Example 22.* Consider a multioutput cell implementing a single-bit full adder. The cell has three inputs: $a$, $b$, and $c_{in}$ and two outputs $sum$ and $c_{out}$. The quotient function for the full adder is shown as a block diagram in

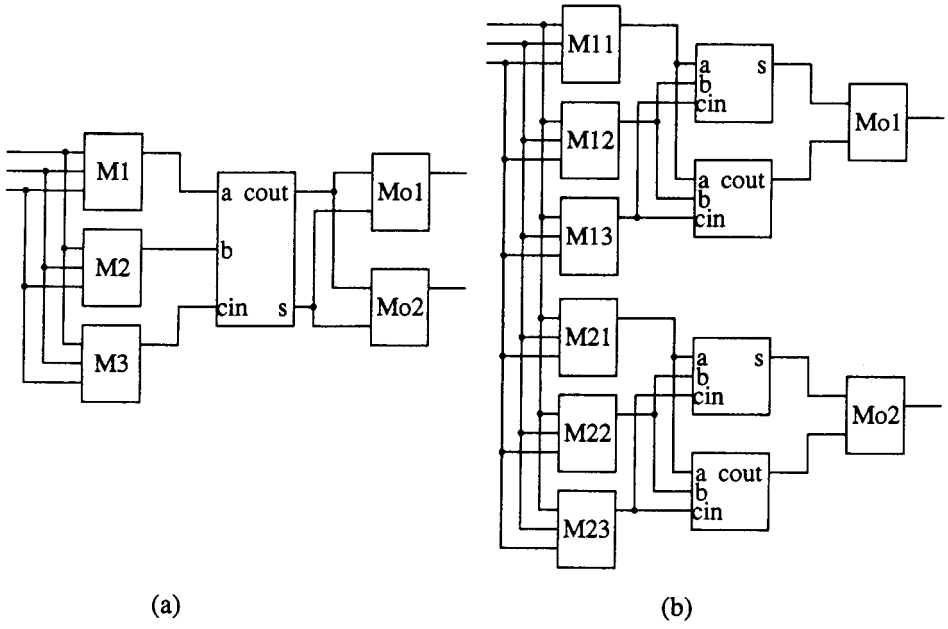


Fig. 12. (a) Generalized matching of a multioutput cell (b) Generalized matching of multiple single-output cells.

Figure 12 (a). Notice that there is one multiplexer for each input variable and one for each output ($N_{libOut} = 2$). The control variables are not shown for the sake of readability. On the other hand, if we were to consider the two single-output pattern functions representing the full adder, we would need two quotient functions (one for each output we want to match) with disjoint control variables. This is shown in Figure 12 (b). Clearly, generalized matching of multioutput cluster function using multioutput cells involves a much smaller number of control variables.

It is a well-known fact that multioutput cells can be beneficial for area, power, and performance [Bolchini et al. 1995]. Unfortunately, multioutput cells have seldom been used in synthesis-based design flows because commercial tools do not exploit them effectively. Generalized matching may obviate this deficiency because it detects the use of multiple-output cells whenever they can be used. Moreover, it is more effective than *ad hoc* techniques that merge cells matched by traditional algorithms because it takes into account the degrees of freedom for multioutput optimization.

## 9. SUMMARY

We have reviewed several techniques for Boolean matching that are applicable to library binding and, in some cases, to combinational logic verification.

We have considered models for matching of increasing complexity. Boolean matching involves both equivalence check and finding an assignment of cluster to pattern variables. Such an assignment may require variable permutation, variable complementation, or may be so general as to allow for combining pattern variables together (input bridging) and/or sticking them to constant values.

Boolean matching is based on exact equivalence-checking techniques that find a match whenever possible. These techniques can be extended to take advantage of the degrees of freedom introduced by *don't care* conditions, as well as those induced by the Boolean relation that models concurrent matching. Both extensions may increase the number of candidate cells matching a given cluster, and thus improve the quality of the resulting mapped network.

Boolean matching is usually implemented by means of operators on BDD representations of the cluster and pattern functions. We have shown how an entire library can be stored in a BDD form and how a matching formula can yield all pattern functions matching a cluster (if any) and the corresponding input assignments.

Finally, we have introduced the concept of generalized matching. With generalized matching, a multiple-output cluster can be matched to two (or more) pattern functions concurrently. We have expressed generalized matching by a Boolean formula that can be implemented by BDDs. In particular, generalized matching can be used to match multiple-output cells.

## ACKNOWLEDGMENTS

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

## REFERENCES

BENINI, L., FAVALLI, M., AND DE MICHELI, G. 1995. Generalized matching, a new approach to concurrent logic optimization and library binding. In *Logic synthesis*.

BRACE, K. S., RUDELL, R. L., AND BRYANT, R. E. 1990. Efficient implementation of a BDD package. In *27th ACM/IEEE design automation conference* (Orlando, FL, June 24-28, 1990). ACM Press, New York, NY, 40–45.

BRAYTON, R., HACHTEL, G., MCMULLEN, C., AND SANGIOVANNI-VINCENTELLI, A. 1984. *Logic minimization algorithms for VLSI synthesis*. Kluwer Academic Publishers, Hingham, MA.

BROWN, F. 1990. *Boolean reasoning*. Kluwer Academic Publishers, Hingham, MA.

BRYANT, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput. C-35*, 8 (Aug.), 677–691.

BURCH, J. R. AND LONG, D. E. 1992. Efficient Boolean function matching. In *Computer-aided design* (Santa Clara, CA, Nov. 8–12, 1992). IEEE Computer Society Press, Los Alamitos, CA, 408–411.

CHEN, K.-C. 1993. Boolean matching based on Boolean unification. In *Computer-aided design*, 346–351.

CHENG, D. I. AND MAREK-SADOWSKA, M. 1993. Verifying equivalence of functions with unknown input correspondence. In *Design Automation*, 81–85.

CLARKE, E. M., MCMILLAN, K. L., ZHAO, X, FUJITA, M., AND YANG, J. 1993. Spectral transforms for large boolean functions with applications to technology mapping. In *Design automation conference* (Dallas, TX, June 14–18, 1993). ACM Press, New York, NY, 54–60.

CONG, J. AND DING, Y. 1992. An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. In *Computer Aided Design*, 48–53.

CONG, J. AND DING, Y. 1996. Combinational logic synthesis for LUT based field programmable gate arrays. *ACM Trans. Des. Autom. Electron. Syst. 1*, 2 (Apr.), 145–204.

DARRINGER, J., JOYNER, W., BERMAN, L., AND TREVILLYAN, L. 1981. LSS: Logic synthesis through local transformations". *IBM J. Res. Dev. 25*, 4 (July), 272–280.

DE MICHELI, G. 1994. *Synthesis and optimization of digital circuits*. McGraw-Hill, Inc., New York, NY.

DETJENS, E. AND GANNOT, G., ET AL. 1987. Technology mapping in MIS. In *Computer-Aided Design*, 116–119.

EDWARDS, C. 1975. Applications of Rademacher-Walsh transform to Boolean function classification and threshold logic synthesis. *IEEE Trans. Comput.* (Jan.), 48–62.

ERCOLANI, S. AND DE MICHELI, G. 1991. Technology mapping for electrically programmable gate arrays. In *ACM/IEEE design automation conference* (San Francisco, CA, June 17–21, 1991). ACM Press, New York, NY, 234–239.

FORTAS, A., BOUZOUZOU, H., CRASTES, M., ROANE, R., AND SAUCIER, G. 1995. Mapping techniques for Quicklogic FPGA. In *SASIMI*.

GAREY, M. AND JOHNSON, D. 1979. *Computers and intractability*. W. H. Freeman & Co., New York, NY.

GREEN, J., HAMDY, E., AND BEAL, S. 1993. Antifuse field programmable gate arrays. *Proc. IEEE 81*, 7 (July), 1041–1056.

GREGORY, D., BARTLETT, K., DE GEUS, A., AND HACHTEL, G. 1986. Socrates: A System for Automatically synthesizing and optimizing combinational logic. In *Design automation*, 79–85.

HURST, S., MILLER, D., AND MUZIO, J. 1985. *Spectral techniques in digital logic*. Academic Press Ltd., London, UK.

KARPLUS, K. 1991. Amap: a technology mapper for selector-based field-programmable gate arrays. In *Design Automation*, 244–247.

KEUTZER, K. 1987. DAGON: technology binding and local optimization by DAG matching. In *Design automation conference* (Miami Beach, FL, June 28-July 1, 1987) A. O'Neill and D. Thomas, Eds. ACM Press, New York, NY, 341–347.

MOHNKE, J. AND MALIK, S. 1993. Permutation and phase independent Boolean comparison. *Integr. VLSI J. 16*, 2 (Dec.), 109–129.

MORRISON, C. R., JACOBY, R. M., AND HACHTEL, G. D. 1989. Techmap: technology mapping with delay and area optimization. In *Logic and Architecture Synthesis for Silicon Compilers*. North-Holland Publishing Co., Amsterdam, The Netherlands, 53-64.

MURGAI, R., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. L. 1992. An improved synthesis algorithm for multiplexor-based PGA's. In *Design automation conference* (Anaheim, CA, June 8-12, 1992). IEEE Computer Society Press, Los Alamitos, CA, 380–386.

SAVOJ, H., SILVA, M. J., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 1992. Boolean matching in logic synthesis. In *European Design Automation* (Congress Centrum Hamburg, Hamburg, Germany, Sept. 7–10, 1992). IEEE Computer Society Press, Los Alamitos, CA, 168–174.

SCHLICHTMANN, U., BRGLEZ, F., AND HERMANN, M. 1992. Characterization of Boolean functions for rapid matching in FPGA technology mapping. In *Design automation conference* (Anaheim, CA, June 8-12, 1992). IEEE Computer Society Press, Los Alamitos, CA, 374–379.

SCHLICHTMANN, U., BRGLEZ, F., AND SCHNEIDER, P. 1993. Efficient Boolean matching based on unique variable ordering. In *Logic Synthesis*.

SOMENZI, F. AND BRAYTON, R. K. 1989. Minimization of Boolean relations. In *Circuits and systems*, 738–743.

TRIMBERGER, S. 1993. A reprogrammable gate array and application. *Proc. IEEE 81*, 7 (July), 1030–1041.

TSAI, C.-C. AND MAREK-SADOWSKA, M. 1994. Boolean matching using generalized Reed-Muller forms. In *Design automation* (San Diego, CA, June 6–10, 1994). ACM Press, New York, NY, 339–344.

WANG, K.-H. AND HWANG, T.-T. 1995. Boolean matching for incompletely specified functions. In *Design automation conference* (San Francisco, CA, June 12–16, 1995). ACM Press, New York, NY, 48–53.

WANG, K. H., HWANG, T.-T., AND CHEN, C. 1996. Exploiting communication complexity in Boolean matching. *IEEE Transactions on CAD/ICAS 15*, 10 (Oct.), 1249–1256.

WATANABE, Y., GUERRA, L. M., AND BRAYTON, R. K. 1996. Permissible functions for multioutput components in combinational logic optimization. *IEEE Transactions on CAD/ICAS 15*, 7 (July), 734-744.

WONG, S., SO, H., OU, J., AND COSTELLO, J. 1989. A 5000-gate CMOS EPLD with multiple logic and interconnect array. In *Custom Integrated Circuit*, 581–584.

YANG, J. AND DE MICHELI, G. 1991. *Spectral techniques for technology mapping: A CSL report*.

YANG, C.-H., CHEN, S.-J., HO, J.-M., AND TSAI, C.-C. 1997. Hmap: a fast mapper for EPGAs using extended GBDD hash tables. *ACM Trans. Des. Autom. Electron. Syst. 2*, 2 (Apr.).

Received January 1997; revised April 1997; accepted April 1997