Cell-based logic optimization

Giovanni De Micheli

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract. This chapter surveys techniques for library binding in semicustom technologies. Library binding is the back-end of logic synthesis, and constructs an interconnection of cell instances from a given library, starting from a multi-level logic network. Emphasis is placed on the algorithmic approach to library binding, with particular reference to covering and matching techniques.

1 Introduction

Computer-aided logic synthesis involves several tasks, most of which covered in reference [18]. Some techniques are considered to be classic, and probably invariant with the improvement of semiconductor technology. On the other hand, the trend toward using deep submicron technologies has renewed the interest in those methods that interface logic design with physical design. For this reason, this survey is centered on those techniques that leverage the use of semicustom design libraries of cells in logic design.

Cell-library binding is the task of transforming a logic network, originally described by a set of logic equations, into an interconnection of components that are cell instances of a given library. Library binding allows us to retarget logic designs to different technologies and implementation styles. For this reason, it is also often called *technology mapping*.

The library contains 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. Typical optimization objectives are either the minimization of the critical delay, or the minimization of power consumption (or area) under delay constraints.

Practical approaches to library binding can be classified into two major groups: *heuristic* algorithms and *rule-based* approaches. In this survey we consider the algorithmic approach and we refer the interested reader to [17,18] for the latter approach.

2 Problem formulation and analysis

The objective of this section is to give a feel for the complexity of the problem and the use of heuristics.

Let us restrict our attention to libraries of combinational single-output cells, each one characterized by its logic function and a single cost parameter (e.g., area). Let us assume we want to determine a cell interconnection yielding a logic network which minimizes the overall cost.

A typical framework for understanding the library binding problem involves the solution to two major tasks.

- *Matching* is determining if a portion of the original logic network is replaceable by a logic cell.
- *Covering* is finding a network interconnection of matching cells, whose overall logic behavior is equivalent to that of the original network.

A cell *matches* a subnetwork when they are functionally equivalent. Thus matching corresponds to solving a tautology problem, which belongs to the Co-NP-complete complexity class [21]. Nevertheless, the usually small number of cell inputs allows us to solve the matching problem efficiently.

The network covering problem is usually much more difficult to solve [18]. Indeed covering entails the selection of an appropriate number of matches, which minimizes the overall associated cost, with the additional requirement that each cell input is connectable to some cell output (or primary input). In other words, the selection of some matches implies the selection of other matches. The covering problem can then be viewed as a *binate covering* problem [18], which is computationally intractable. Unfortunately, the large number of matches in usual networks, and the binate nature of the problem, make it unlikely to be solvable exactly on present-day computers.

Example 1. Consider the simple library shown in Figure 1 (a) and the unbound network of Figure 1 (b) [18]. We consider the problem of finding a network cover that minimizes the total area cost associated with the chosen cells.

There are several possibilities for covering this network. For example, a trivial binding is shown in Figure 1 (c). A more interesting binding can be found by considering the possible matches. Consider, for example, vertex v_1 that can be bound to a two-input OR gate (OR2), and vertex v_2 that can be bound to a two-input AND gate (AND2). Moreover, the subnetwork consisting of $\{v_1, v_2\}$ can be bound to a complex gate (OA21). We can associate binary variables to denote the matches. Variable m_1 is TRUE whenever the OR2 gate is bound to v_1, variable m_2 is TRUE whenever the AND2 gate is bound to v_2, and m_4 represents the use of OA21 for covering the subnetwork $\{v_1, v_2\}$. Similar considerations apply to vertex v_3. We use variables m_3 to denote the choice of a two-input AND gate (AND2) for v_3, and m_5 to represent the choice of OA21 for $\{v_1, v_3\}$ respectively. The possible matches are shown in Figure 1 (d).

Therefore we can represent the requirement that v_1 be covered by at least one gate in the library by the unate clause $(m_1 + m_4 + m_5)$. Similarly, the

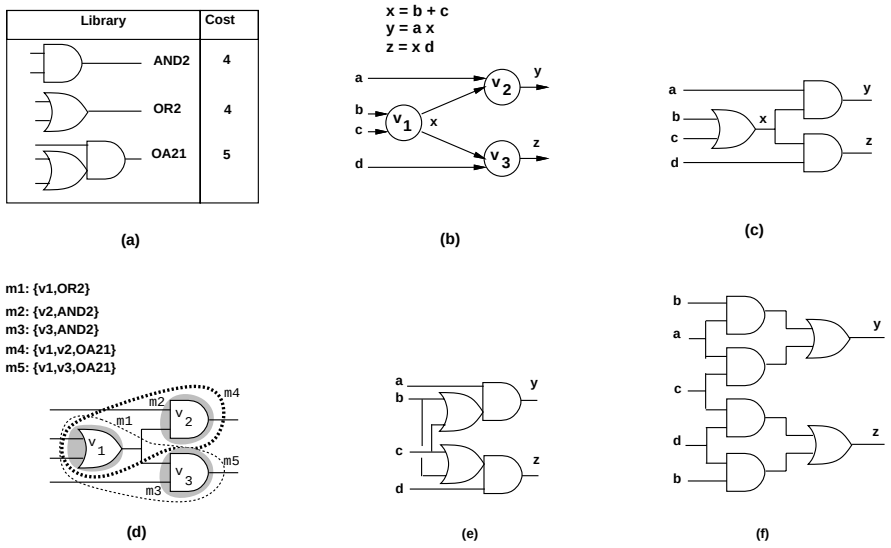

Fig. 1. (a) Simple library. (b) Unbound network. (c) Trivial binding. (d) Match set. (e) Network cover. (f) Alternative bound network which is not a cover of the unbound network shown in (b).

covering requirements of v_2 and v_3 can be represented by clauses $(m_2 + m_4)$ and $(m_3 + m_5)$ respectively.

In addition to these requirements, we must insure that the appropriate inputs are available to each chosen cell. For instance, binding an AND2 gate to v_2 requires that its inputs are available, which is the case only when an OR2 gate is bound to v_1. The former choice is represented by m_2 and the latter by m_1. This implication can be expressed by $m_2 \rightarrow m_1$, or alternatively by the binate clause $(m_2' + m_1)$. Similarly, it can be shown that $m_3 \rightarrow m_1$, or $(m_3' + m_1)$. Therefore, the following overall clause must hold:

$$(m_1 + m_4 + m_5)(m_2 + m_4)(m_3 + m_5)(m_2' + m_1)(m_3' + m_1) \ = \ 1$$

The clause is binate. An exact solution can be obtained by binate covering, taking into account the cell costs. For each cube satisfying the clause, the least-cost one denotes the desired binding. In this case, the optimum solution is represented by cube $m_1' m_2' m_3' m_4 m_5$, with a total cost of 10, corresponding to the use of two OA21 gates. The optimal bound network is shown in Figure 1 (e).

The bound network obtained by covering depends on the initial decomposition. For example, Figure 1 (f) shows another bound network, which is equivalent to the network of Figure 1 (b), but not one of its covers.

3 Algorithms for library binding

Heuristic algorithms have been developed for the library binding problem, due to the intrinsic difficulties of solving exactly even simple library binding problem and to the practical nature of realistic binding problems, which involve more complex library cells and cost models.

Algorithms for library binding were pioneered by Keutzer at AT&T Bell Laboratories [25], who recognized the similarity between the library binding problem and the code-generation task in a software compiler. In both cases, a matching problem addresses the identification of the possible substitutions, and a covering problem the optimal selection of matches.

There are two major approaches to solving the matching problem, which relate to the representation being used for the network and the library. In the *Boolean* approach, the library cells and the portion of the network of interest are described by Boolean functions and matching is formulated as solving a tautology problem [32]. In the *structural* approach, graphs representing algebraic decompositions of the Boolean functions are used instead. Matching is thus reduced to a (sub-)isomorphism problem [19,25,38].

Typical objective and constraint functions in mapping are *area, delay* and *power.* Computing the area cost of a mapped network is usually straightforward, because the area cost of the cell is the sum of the cell areas. Note that precise area estimate should include inter-cell routing space. Delay computation is more involved, because the critical path(s) must be traced. Cell delays

depend on the loading factor, and thus the impact of the choice of a cell on the critical path delay needs estimation of the driven loads. When mapping a network by traversing it from its primary inputs to its outputs, the cell delay evaluation may require a look-ahead step. Moreover, accurate delay estimation requires the detection of *false paths* [18] and their elimination from consideration.

Power estimation is even more involved. The cell's power consumption depends on its output capacitive load, on the output switching activity [36] and on the input patterns. In the presence of feedback (i.e., when binding the combinational portion of a sequential network), spatio-temporal correlations complicate the computation of the switching activities. Thus a variety of methods [33,36] are used to estimate the power consumption tied to the choice of a cell.

Because of the difficulty of estimating accurately the delay and power cost functions, *re-mapping* techniques have been proposed that iteratively improve a mapped network [11,14,27,37,44,45]. In re-mapping, the initial network costs in terms of area, delay and power are known, and the impact of physical design on these costs may also be known (by measuring the wiring capacitances). Re-mapping techniques use a peephole approach that highlights a portion of the network where the existing cell binding is altered to improve the cost. The peephole is then moved over another part of the network.

3.1 The classical approach to library binding

The major difficulty in solving the library binding problem lies in the network covering problem, as mentioned before. To render the problem solvable and tractable, most heuristic algorithms apply two pre-processing steps to the network before covering: *decomposition* and *partitioning*.

Decomposition is required to guarantee a solution to the network covering problem, by insuring that each vertex is covered by at least one match. The goal of decomposition in this context is to express all local functions as simple functions, such as two-input NORs or NANDs , that are called *base functions*. The library must include cells implementing the base functions, to insure the existence of a solution. Indeed, a trivial binding can always be derived from a network decomposed into base functions.

Different heuristic decomposition algorithms can be used for this purpose, but attention must be paid for because network decompositions into base functions are not unique and affect the quality of the solution. Therefore heuristics may be used to bias some features of decomposed networks. For example, while searching for a minimal delay binding, a decomposition may be chosen such that late inputs traverse fewer stages.

The second major pre-processing step in heuristic binding is partitioning, that allows the covering algorithm to consider a collection of multiple-input single-output networks in place of a multiple-input multiple-output network.

The subnetworks that are obtained by partitioning the original network are called *subject graphs* [25]. Subject graphs are then covered by library elements one at a time. Networks are usually partitioned at multiple-fanout points.

Finally each subject graph is covered by an interconnection of library cells, as described next.

Tree-covering. Tree-covering is based on a structural representation of the cells and of the subject graph by means of trees. Usually, decomposition into base functions leads to *directed acyclic graph* (dag) representations of cells and subject graphs, which can be split to obtain trees where matching and covering can be carried out more efficiently.

Optimum tree covering can be computed by dynamic programming [3,25].

We describe here briefly a minimum-area covering problem and solution. Each cell has a tree pattern and an area cost. The tree covering algorithm traverses the subject graph in a bottom-up fashion.

For all vertices of the subject tree, the covering algorithm determines the matches of the locally rooted subtrees with the pattern trees. There are three possibilities for any given pattern tree.

1. The pattern tree and the locally rooted subject subtree are isomorphic. Then, the vertex is labeled with the corresponding cell cost.
2. The pattern tree is isomorphic to a subtree of the locally rooted subject subtree with the same root and a set of leaves L. Then, the vertex is labeled with the corresponding cell cost plus the labels of the vertices L.
3. There is no match.

If we assume that the library contains the gates implementing the base functions, then for any vertex there exists at least one cell for which one of the first two cases applies, and we can label that vertex. Therefore, it is possible to choose for each vertex in the subject graph the best labeling among all possible matches. At the end of the tree traversal, the vertex labeling corresponds to an optimum covering. Note that overall optimality is weakened by the fact that the total area of a bound network depends also on the partitioning and decomposition steps. The complexity of the algorithm is linear in the size of the subject tree.

Example 2. Consider the network shown in Figure 2, its subject graph and a library with cells {INV ,NAND2 ,AND2 ,OR2 } having area costs 2,3,4,5 respectively.

The bottom-up application of the tree-covering algorithm is shown in Figure 3. At the first step, nodes s and u are matched with an INV and NAND2 respectively. Next, the algorithm attempts to match node t with an INV and with an AND2 . Such matches yield costs of 5 and 4 respectively, and so node t is matched with an AND2 . Last, the algorithm attempts to match node r with a NAND2 and an OR2 . These matches yield costs of 9

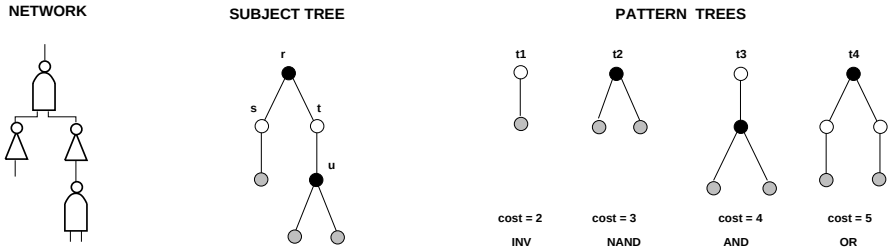

Fig. 2. Network, subject tree, and pattern trees

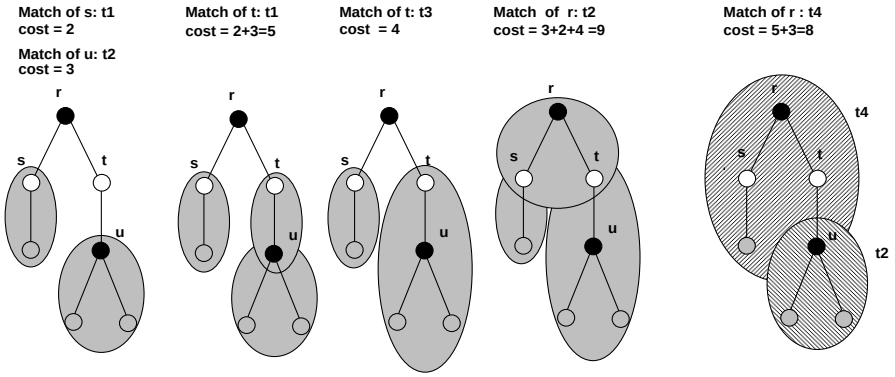

Fig. 3. Example of covering by dynamic programing

and 8 respectively, and so the best area cover involves an OR2 gate fed by a NAND2 .

Tree-covering can be extended to cope with delay minimization. If the gate delay is constant, then it suffices to compute the cost at each node as the sum of the gate delay plus the largest cost of its children (i.e., largest input arrival time). When gate delay is load-dependent, the algorithm can be modified by using binning techniques [18,38]. Most covering methods, targeting different cost functions and models, use a framework similar to Keutzer's algorithms.

3.2 Limitations and extensions of the classical approach

The classical approach of Section 3.1 has been the basis for several research and commercial implementations. Improvements on the basic technique affect how partitioning, decomposition and matching are done, and will be described in the sequel. Such improvements extend the search space and thus allow the algorithms to find higher-quality solutions. At the same time, they attempt to provide better estimation of the cost functions of interest. For example, a combined approach to solving cell binding and placement has been proposed recently [31], where the overall area (due to cells and to routing) can be estimated and optimized.

Avoiding partitioning. Circuit partitioning may provide sub-optimal solutions, due to the discontinuity of mapping at the multiple-fanout points. This problem was realized early on [19]. A simple but effective solution is to avoid partitioning, and to start binding at a primary output, and continue along the fanin cone until the algorithm reaches the primary inputs or the outputs of a previously-bound cell. Since the overall solution depends on the order in which the primary outputs are considered, heuristic rules can be used, such as selecting first the most timing-critical outputs.

When considering mapping for minimum delay with a constant delay model, an optimum solution can be found using an unpartitioned multiple-output dag. In this case, a combinational network can be mapped as a whole, but the final result still depends on the chosen decomposition. This result was first shown by Cong and Ding [16] while considering libraries represented by look-up tables (LUTs) with a bounded number of inputs, as in the case of field-programmable gate arrays (FPGAs). Their method relied on a network traversal in topological order, with an optimum vertex labeling which satisfies the principle of optimality of dynamic programming. Later, Kukimoto et al. [26] showed that this result is applicable to generic libraries, by using a similar network traversal and an extended matching concept, which allows nodes to be covered by more than one cell (i.e., by allowing duplication).

Representing all possible decompositions. Lehman et al. [28] addressed the problem of optimizing the mapped network over multiple algebraic decompositions. The network is modeled by a *mapping graph*, which is a network representation by means of AND2 and INV base functions, with two modifications. First, *choice nodes* are introduced. They are "virtual multiplexers" that can feed any newtork node with the output of two nodes representing alternative decompositions of the corresponding function. Second, the mapping graph can contain directed cycles. For example, a cycle consisting of two inverters and two choice nodes can indicate that a function f may be expressed as the complement of a function g and vice versa (See Figure 4). Restrictions are applied to the use of cycles, to insure uniqueness of the function assigned to the network vertices.

Mapping graphs can be *reduced* by removing redundancies. In a reduced mapping graph, there are no distinct choice nodes with logically-equivalent outputs, and there are neither AND2 nor INV nodes with identical inputs.

Example 3. Consider the two equivalent AND2 / INV networks of Figure 4 (a) [28]. They can be represented as a single mapping graph, by using a choice node, as shown in Figure 4 (b). Next, the graph can be reduced, to remove redundancies. First, nodes 6 and g are made input of a common choice node, because they are logically equivalent. Then, 7 and h are merged because they have an identical input. Next, 7, h and f are made inputs of a common choice node, because they are are equivalent, creating a cycle. Finally, several nodes are merged because they have identical inputs: 3 and c, 8 and i, 9 and j. The cycle indicates that any even number of inverters may appear between f and 8,i, and that any odd number of inverters may appear between 6 and 8,i.

Lehman et al. proposed an algorithm for traversing the mapping graph and matching cells to nodes of this graph, using a specialed graph matching technique. The covering algorithm yields the best mapped network, among those that can be derived from the decompositions encoded by the mapping graph. Since the mapping graph does not represent all possible decompositions, the mapping graph is iteratively modified during covering using associative, distributive and inverter (addition and removal of inverter pairs) transformations [28]. The result of applying these transformations yields a method that is at least as powerful as exhaustively applying algebraic decomposition. While the interested reader is referred to [28] for details, it is important to stress that the implementation of this method runs fast and is widely applicable to both area and delay minimization.

Limitations of structural matching. Representations of logic functions by algebraic decompositions into base functions are trees or dags. (For example an EXOR2 function can be represented by a dag with NAND2 and INV base functions). While tree matching can be done efficiently in liner time,

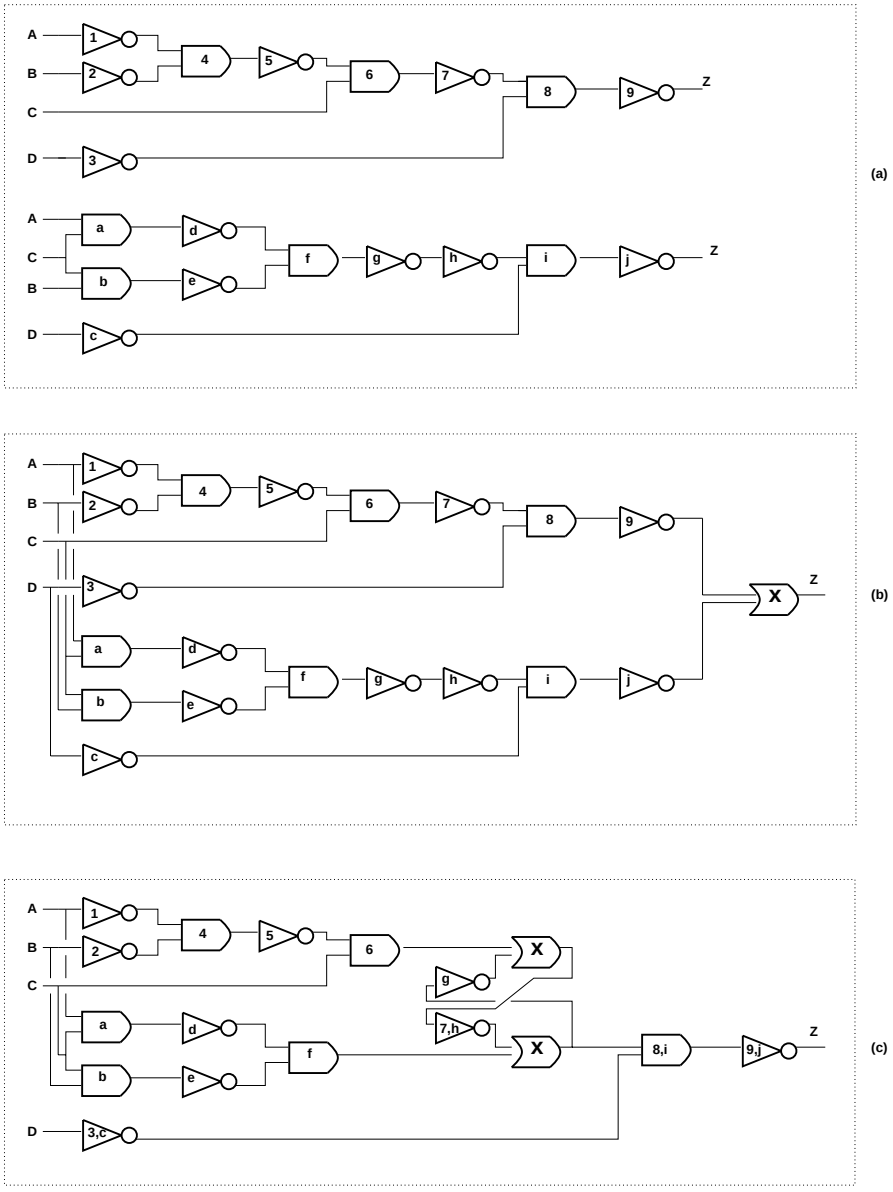

Fig. 4. (a) Two logically-equivalent network decompositions. (b) Mapping graph showing how a choice node (marked by an X) encodes the two decompositions. (c) Reduced mapping graph.

there are some pitfalls in using algebraic representations and the corresponding structural matching techniques. First, algebraic decompositions are not canonical, and thus a possible match may be missed if a library cell has a decomposition different from that of the network portion to which it is compared. This problem can be alleviated by using multiple decompositions, as mentioned in the previous section. Second, cells that are represented by dags (instead of trees) require more complex matching algorithms. Third, algebraic decompositions do not capture *don't care* conditions, which are useful in determining lower-cost solutions.

For these reasons, Boolean matching techniques have been used to solve the tautology problems, by leveraging Binary Decision Diagrams [9] which are canonical representations for logic functions. Such techniques have been shown to have run-times competitive with structural matching methods. Usually, Boolean matching is applied within a general framework where a network is decomposed into base functions beforehand. As with structural covering, the library is required to include the base functions. At each network node, it is then possible to form clusters (by aggregating base functions) which are characterized by *cluster functions*. Covering can then be performed bottom-up, by applying Boolean matching to the cluster functions and by combining the cost of the match with the cost of the covers of the subgraphs rooted at the vertices providing the inputs to the matching cell. This procedure can be used to minimize arbitrary objective functions. Boolean matching techniques are extensively described in Section 4.

3.3 Re-mapping

The quality of bound networks (i.e., their area, delay and power consumption) depends much on parameter estimation. A good (even an optimum) algorithm is useless if the input data is inaccurate. When considering submicron technologies, delay (but also area and power consumption) depends mainly on the wiring among cells. Thus estimation of wiring delays is extremely important. Since wiring depends on the placement and on the cell interconnections, two approaches are feasible: i) incorporating physical-design estimators within logic synthesis and library binding tools; ii) performing mapping after physical design. Whereas the latter approach is prone to many far-fetched interpretations, we consider here the re-mapping approach, which is already used in some design flows. In this case, a logic network is mapped first and then cells are placed and wired. After delay extraction, re-mapping is applied to improve the quality of the network.

Several algorithms have been developed that operate on a mapped netlist and attempt to further optimize it [14,27,37,44]. Some re-mapping approaches [11] focus on changing the connectivity of the netlist in such a way that some gates either become redundant (and can be removed) or become sub-optimal (and can be replaced). Re-mapping transformations based on changes of the network connectivity are often called *re-wiring*.

In re-mapping, delay (as well as power and area) estimation drives the selection of the regions to be improved. A logic representation of these clusters is extracted, and then clusters are re-mapped. The newly bound cluster replaces the target region if an improvement is achieved for the cost functions of interest. The process is iterated until the desired improvement is reached or no improvement is possible.

Two techniques are important in re-mapping: forming the clusters and their mapping. Since clusters may have multiple outputs, specialized matching algorithms are required. We describe Boolean matching for multiple-output clusters in Section 5.

The construction of the clusters may be guided by different principles. One approach [45] focuses on selecting multiple-fanout points, and re-mapping a cluster around these points. Multiple-fanout points have usually higher capacitive loads, and thus may be part of critical paths. Once a cluster is chosen, a neighborhood around the cluster has to be determined as well. The neighborhood encapsulates the part of the Boolean network, whose logic behavior expresses the constraints and the degrees of freedom for matching the cluster. Note that for multiple-output clusters, the degrees of freedom for matching are expressed by Boolean relations [42], as shown in Section 5.

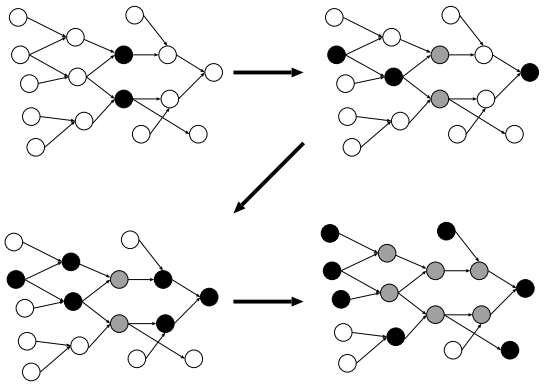

Fig. 5. Building the neighborhood of a cluster

Example 4. We show on Figure 5 a method for constructing a neighborhood [45]. The picture represents a portion of a logic network, the nodes being logic gates and the arrows the connections between them. We start from a two-node cluster, marked in black in the top left part of the Figure 5. To build the neighborhood, we first select the reconvergent nodes in the transitive fanout and fanin of the clusters (with depth 2) from the cluster. These nodes are marked in black on the top right. The nodes on paths connecting the cluster with reconvergent nodes are marked in black on the bottom left.

Finally, we take the "envelope" of these nodes to get the neighborhood. The neighborhood is the set of nodes marked in black in the bottom right part of Figure 5.

Overall the advantages of re-mapping are: i) it merges seamlessly with pre-existing tools and design flows, ii) it allows us to put more effort in local optimizations. The main drawback is that re-mapping performs only incremental improvements, thus, if the starting point is a local optimum very far from the global optimum, we may not be able to move out of it.

4 Boolean matching

Whereas tree-based matching is a subject broadly described in textbooks [1], information on Boolean matching is scattered in the technical literature. Because of the increasing application of Boolean matching techniques in library binding tools, we dedicate the next two sections to a detailed survey of Boolean matching.

We consider first Boolean matching for single-output cluster functions. Matching incompletely-specified functions will be described in Section 4.3 and matching multiple-output cluster functions in Section 5.

4.1 Preliminaries

We model combinational clusters and library cells by *cluster functions* and *pattern functions* respectively. For now, we assume both functions to have multiple inputs and a single output. We denote vectors and matrices in boldface, and the vector whose entries are 1 by $\mathbf{1}$.

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 $n = m$ unless specified otherwise. We shall remove this assumption in Section 4.4. 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. 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. Then we remove this restriction and formulate the complete Boolean matching problem.

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 [6].

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 [32] 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 [6].) To expedite $\mathcal{P}$-equivalence checks, he used filters to prune unnecessary tautology checks (See Section 4.2.) The method can be perfected by associating each library element with a multi-rooted ROBDD representing all variable permutations.

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 $f(\mathbf{x}) \overline{\oplus} \mathcal{N}_o g(\mathcal{P}\ \mathcal{N}_i\ \mathbf{x})$ is a tautology [24]. 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 $f(\mathbf{x}) \overline{\oplus} g(\mathcal{N}_i\ \mathbf{x})$ 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 reduce drastically the number of tries, and early approaches to Boolean matching were relying heavily on filtering [32]. Moreover, canonical forms can be used to check for equivalence in constant time.

Variable assignment and Boolean matching. In practice, a cluster function is defined over some network variables $\mathbf{x}$ and a pattern function is defined

over some other variables $\mathbf{y}$. 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 $\mathbf{x}$ into $\mathbf{y}$.

Example 5. 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 $\mathbf{y} \overline{\oplus} \mathbf{PN} \oplus \mathbf{x} = \mathbf{1}$, where $\mathbf{N}$ is a diagonal Boolean matrix.

The pattern function g under the variable assignment represented by $\mathcal{A}$ is [39]:

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

Example 6. 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 6.)

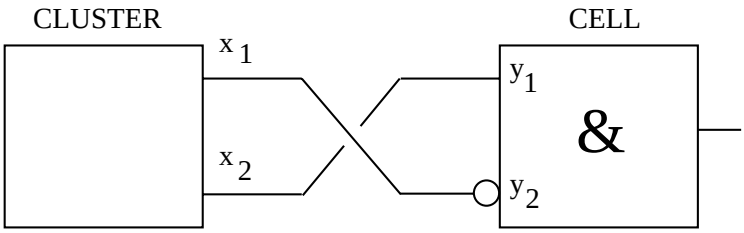

Fig. 6. Input assignment in matching

Let us consider $\mathcal{PN}$-equivalence, being the extension to $\mathcal{NPN}$-equivalence straightforward. 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 $\mathbf{x}$. Therefore there is a Boolean matching if and only if the following formula evaluates to true.

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

4.2 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.

Canonical forms. Burch and Long introduced a canonical form for representing functions modulo input-polarity assignments [10]. 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, then f is $\mathcal{N}$-equivalent to $\mathcal{C}_\mathcal{N}(f)$. Moreover, if f is $\mathcal{N}$-equivalent to g, then $\mathcal{C}_\mathcal{N}(f) = \mathcal{C}_\mathcal{N}(g)$.

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 [6,18]. A description is reported in [10].

Let us consider now 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 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 computational speed reasons, Burch and Long [10] proposed the use *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 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, by having the library hash table store the permutation semi-canonical forms in polarity canonical form. Finally, checking for $\mathcal{NPN}$-equivalence is usually done by checking also for $\mathcal{PN}$-equivalence of the complement of f.

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 [34].

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 4.2.

Mailhot [32] 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 variable assignment 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 (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 also 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 7. 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 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 [32,35]. 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 8. 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: $\{x_1x_2x_3\}$, $\{x_4x_5\}$, and $\{x_6x_7\}$. 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 [9]. 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 [34] suggested to consider the satisfy counts of the co-factors 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 [9]. 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 co-factors are the same. Mohnke and Malik [34] 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 reference [34].

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* [34]. 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, Sastry and Pedram [30], 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, Hwang and Chen [47] 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. π. Then, given a function f and a variable bipartition π, the equivalence signature is the set of k pairs, each defined by: i) the satisfy count of an equivalence class and ii) the co-factor 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 [47] 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 in library binding, because it can handle functions with more variables (e.g., 10-100) than other methods.

Schlichtmann, Brglez and Herrmann [40] 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 path with even or odd parity) and those patterns that inhibit the fault sensitization. Cheng and Marek-Sadowska [13] used signatures based on *partner patterns* and *cofactor statistics*, which can be reconduced to single-fault propagation signatures by scaling and by modifying the format.

Finally, Tsai and Marek-Sadowska [43] 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.

Spectral methods. There are several spectral representation of Boolean functions [24]. 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:

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

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 $\mathbf{s}$ 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 9. Consider $f(x_1, x_2, x_3) = x_1 x_2 + x_3'$ $(n = 3)$. 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 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) 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 [20], 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 [49] 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, 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 the spectrum of 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. [15] 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 [15] applied spectral filters to speed-up matching, and gave experimental evidence on the high *effect/cost ratio* of such filters.

4.3 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 [18] for the computation of f_{DC}. We say that a pattern function g matches a cluster function f, if g matches $\widetilde{f}$ where $f \cdot f'_{DC} \leq \widetilde{f} \leq f + f_{DC}$.

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 [32]. 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 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 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.

Example 10. Consider the matching compatibility graph of Figure 7, 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 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 of 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 $ab + b'c$.

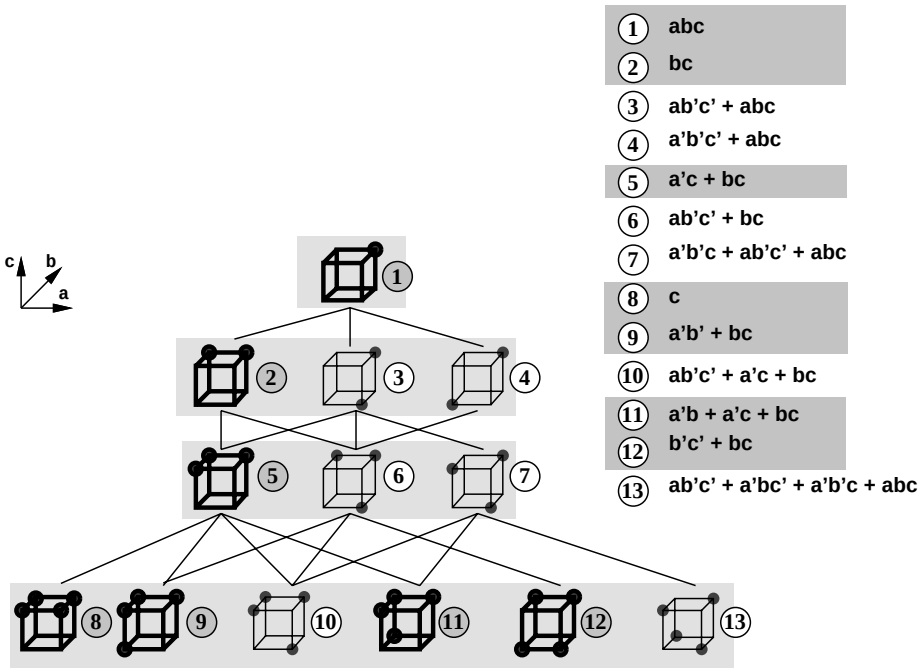

Fig. 7. Matching compatibility graph with library annotation.

A formula for Boolean matching with *don't care* conditions. Savoj et. al [39] 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:

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

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 11. 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 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 $\forall_{\mathbf{x}}(\exists_{\mathbf{y}}((y_1 \oplus x_1)(y_2 \overline{\oplus} x_2)(x_1' x_2 + (x_1 \ \overline{\oplus} \ x_2) \overline{\oplus} (y_1 + y_2))))$. 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 (4) is satisfied. (Figure 8).

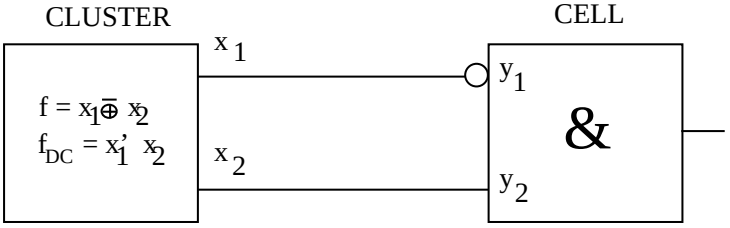

Fig. 8. Input assignment in matching with *don't care* conditions

The main problem in using Formulae (3) and (4) is to find the variable assignment. Savoj et. al ([39]) 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 [39] for details.

Boolean unification. *Boolean unification* is the process of finding a solution of a Boolean equation [8]. A method for finding Boolean matching with *don't care* conditions based on Boolean unification was proposed by Chen [12]. 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 [12] 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 $\mathbf{N}$. 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 [12] 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 [12] does not represent a significant improvement upon enumerative procedures enhanced by efficient filters.

Matching using multi-valued functions. One recent approach to Boolean matching with *don't care* [46] 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 12. 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 sum of products representations

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. Then, the pattern functions are complemented, and stored also in *sum of product* form. We consider a cluster function f 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 [46].

Example 13. 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 in library binding, 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 [7]. Moreover, for most libraries, the sum of cubes 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 purposes, the larger number of inputs can lead to situations where the size of the *sum of products* 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 [46] 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.

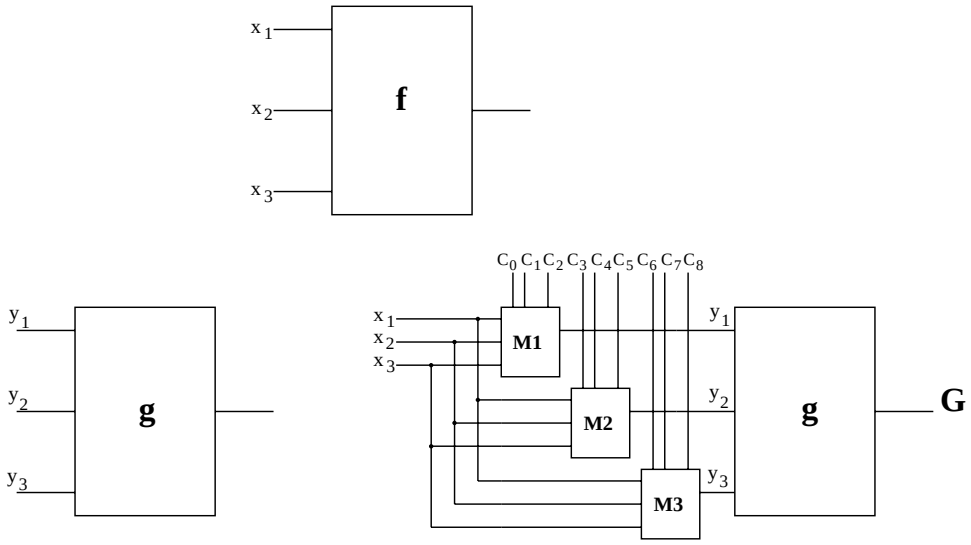

Fig. 9. *Transformation of the pattern function g into G for matching with cluster function f. The first two control variables of each multiplexer are for permutation control, the last one is for polarity control.*

4.4 Library matching

We generalize now the concept of Boolean matching, with the goal of being able to compare a cluster function with a data structure representing all libray cells and allowing for any pin assignment and complementation. Therefore we will extend the concept of matching step by step in this section.

First, 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 EXOR 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 9.

Example 14. Consider box M1 in Figure 9, 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 inverting, 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 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 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 that 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 $G(\mathbf{c}, \mathbf{x})$ *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. Intuitively, the formula can be explained by observing that there is an $\mathcal{EPN}$ matching if and only if there exists an assignment $\mathbf{c}^*$ 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 variables $\mathbf{c}$. We shall call it *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.

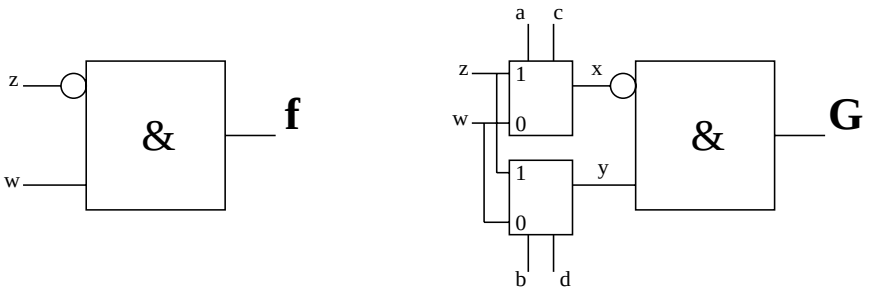

Fig. 10. *Pattern function f and quotient function G of Example 3.*

Example 15. Let the pattern function be $g = x'y$ and the cluster function be $f = wz'$. Figure 10 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:

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

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 $M(a,b,c,d)$ 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 we want to keep the number of variables in the ROBDD representation of G as small as possible for efficiency reasons.

When the cluster function is completely specified, traditional matching procedures enhanced with filters appear to be more efficient than using the quotient function, because the tautology check is fast and the number of checks is reduced to one (or few) in most practical cases [41]. However, our approach is applicable to much more general Boolean matching problems, where traditional techniques cannot be applied. We shall 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:

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

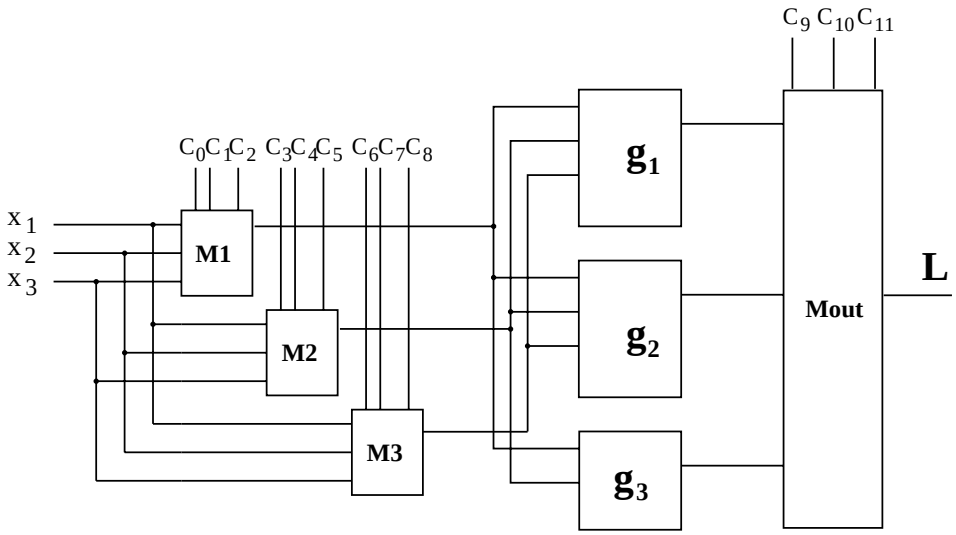

Fig. 11. Quotient function for cell selection and matching

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 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.

We describe now a further extension of the matching formulation, that allows us to combine matching and cell selection in a single step. We call it *library matching.* This extension is important because the generalized formula denotes all cells and corresponding variable assignments which 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.

Library matching 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 11. In addition to input multiplexing and complementation, also the cell outputs are 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 outpus, c_{11} selects the polarity of the connection.

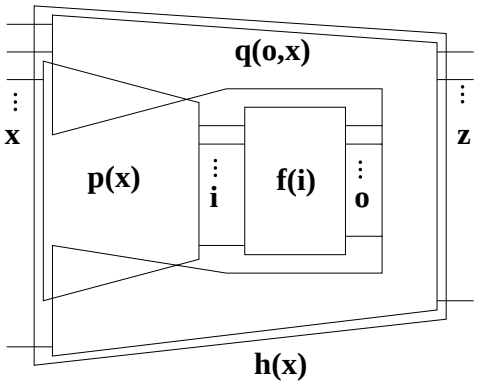

Fig. 12. A multi-output cluster function embedded in its environment

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 for which library cell the input assignment leads to 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 11. 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. Cell 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 $c_{11} = 1$, non-inverting 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 11) input, x_2 connected to its second input and x_3 connected to its third input. No input is inverted.

5 Generalized matching

We remove now the restriction of dealing with single-output clusters and cells. We extend our approach to cope with matching concurrently the multiple outputs of a cluster, and we call it *generalized matching*. We describe the approach of Benini, Vuillod et al. [44,45] who tackled this problem first. 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 12. We have a multi-output 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 cares* 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 to find a group of single-output pattern functions that satisfy a constraint expressed as a *Boolean relation* [42]. In the following, we adopt a formalism similar to that used by Watanabe et al. [48] in their work on multi-output 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 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:

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

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

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

The characteristic functions fully describe the environment around the multi-output function $\mathbf{f}$. In particular, they enable the computation of a Boolean relation representing the complete set of *compatible functions* of $\mathbf{f}$, i.e., functions that can implement $\mathbf{f}$ without changing the input-output

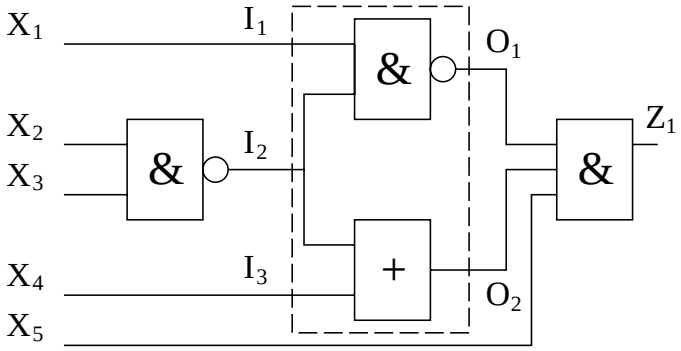

Fig. 13. *A two-output cluster function embedded in a Boolean network*

behavior of **h**. Watanabe et al. showed that the characteristic function $\mathcal{F}$ of the Boolean relation can be obtained by the following formula [48]:

$$\mathcal{F}(\mathbf{i},\mathbf{o}) = \forall_{\mathbf{x},\mathbf{z}}[(P(\mathbf{x},\mathbf{i}) \cdot Q(\mathbf{o},\mathbf{x},\mathbf{z})) \Rightarrow H(\mathbf{z},\mathbf{x})] \tag{11}$$

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 (11) 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 relation.

Example 18. Consider the Boolean network shown in Figure 13. The dashed rectangle encloses the multi-output 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 three outputs: $p_1 = x_1$, $p_2 = (x_2 x_3)'$ and $p_3 = x_4$.

Applying Equation (11) we obtain the Boolean relation representing all degrees of freedom in the implementation of **f**. For ease of understanding, it is given in tabular form:

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

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 (11), we can derive the generalized matching equation. Assume that the multi-output cluster function $\mathbf{f}$ has n_o outputs. We call $\mathcal{L}_k$ the characteristic functions of n_o quotient functions, one for each output of the multi-output cluster function $\mathbf{f}$. 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:

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

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 $\mathbf{o}^* = [o_1^*, o_2^*, ..., o_{n_o}^*]^T$, the quotient functions associated with each component assume a consistent value: $L_1 \equiv o_1^*, \ L_2 \equiv o_2^*, ..., L_{n_o} \equiv o_{n_o}^*$. The universal quantifier on the inputs $\mathbf{i}$ 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 $\mathbf{c}$ on the left-hand side of Equation (12) is the concatenation of the control variables of all n_o quotient functions: $\mathbf{c} = [\mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_{n_o}]^T$. The ON-set of $M(\mathbf{c})$ includes all configuration of control variables representing the ways in which the library cells can be connected so as to obtain a final implementation of $\mathbf{f}$ contained in relation $\mathcal{F}$.

Example 19. Consider the two-output, three inputs cluster function $\mathbf{f}$ 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 $L_1(i_1, i_2, i_3, c_0, ..., c_{10}, c_{11})$ and $L_2(i_1, i_2, i_3, c_{12}, ..., c_{23})$. Notice that L_1 and L_2 have different support, but are otherwise identical. The characteristic functions of the quotient functions are: $\mathcal{L}_1(i_1, i_2, i_3, o_1, c_0, ..., c_{11}) = L_1 \overline{\oplus} o_1$ and $\mathcal{L}_2(i_1, i_2, i_3, o_2, c_{12}, ..., c_{23}) = L_2 \overline{\oplus} o_2$.

The generalized matching equation is:
$M(c_0, ..., c_{21}) = \forall_{i_1, i_2, i_3} \exists_{o_1, o_2} (\mathcal{F}(i_1, i_2, i_3, o_1, o_2) \cdot \mathcal{L}_1(i_1, i_2, i_3, o_1, c_0, ..., c_{11}) \cdot \mathcal{L}_2(i_1, i_2, i_3, o_2, c_{12}, ..., c_{23}))$ where $\mathcal{F}(i_1, i_2, i_3, o_1, o_2)$ is the characteristic function of the Boolean relation for $\mathbf{f}$ computed in Example 18. A minterm $\mathbf{c}^*$ of M uniquely identifies two library cells and an input assignment.

Generalized matching is performed by directly implementing Equation (12) using standard BDD operators. The number of control variables in Equation (12) 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 $\mathbf{f}$ and m_{max} is the maximum number of inputs of a library cell. The term multiplied by n_o is the

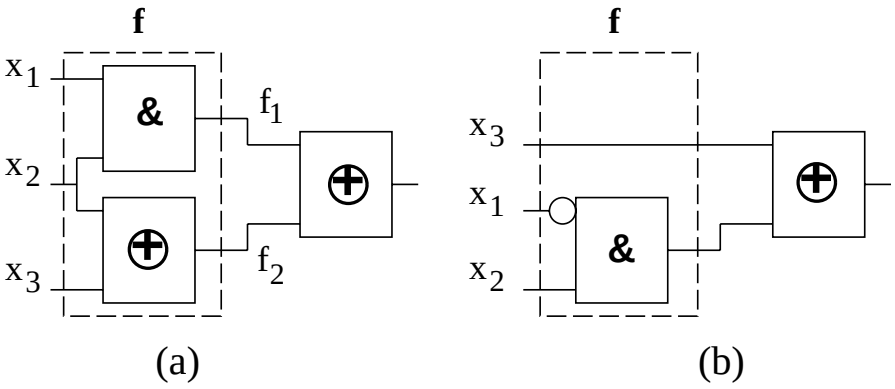

Fig. 14. An example of the effectiveness of generalized matching.

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, the linear contribution is for input polarity assignment.

Example 20. Referring to the multi-output 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. Results are reported in References [44,45]. The enhanced power of generalized matching will be clarified through an example.

Example 21. 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 $g = in_1' in_2$, $Cost = 3$). An implicit cell

is the "WIRE" (cost zero). We want to optimize the mapped network of Figure 14 (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 multi-output 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 14 (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 multi-output library cells. The second and third term account for the input permutations and polarity assignments.

Example 22. Consider a multi-output 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 Figure 15 (a). Notice that there is one multiplexer for each input variable and one for each outputs ($N_{libOut} = 2$). The control variables are not shown for simplicity.

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 15 (b). Generalized matching of multi-output cluster function using multi-output cells involves a much smaller number of control variables.

It is a well-known fact that multi-output cells can be beneficial for area, power and performance [5]. Unfortunately multi-output cells have seldom been used in synthesis-based design flows because commercial tools do not exploit them effectively. Generalized matching may obviate to 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

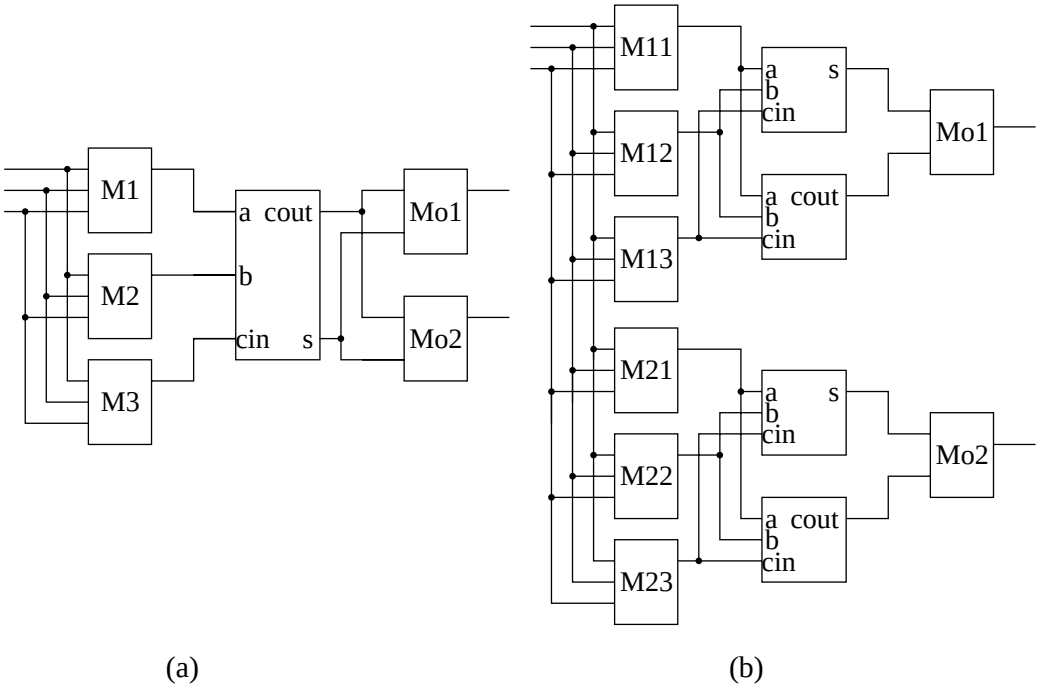

Fig. 15. (a)Generalized matching of a multi-output cell (b) Generalized matching of multiple single-output cells.

available for multi-output optimization. Overall, generalized matching finds its best application with the frame of re-mapping algorithms.

6 Conclusion

Library binding is an important task in logic synthesis, and it provides the bridge between technologically-independent logic networks and netlists of cells to be placed and wired. Whereas rule-based systems played a role in the early development of tools for library binding, most recent approaches exploit algorithms. Despite the fact that some subproblems can be solved exactly and efficiently, heuristics are used to guide the overall mapping process.

Optimization of delay, power consumption and area is performed concurrently with library binding, because the selection of each cell affects the overall quality of the network. With the advent of deep sub-micron technologies, where interconnect delay dominates, the quality of these optimizations depends critically on interconnect estimation. For this reason, future trends will involve both the iteration of binding and physical design (e.g., re-mapping) as well as the merging of these two design phases.

Acknowledgments

The Author acknowledges the scientific contribution of Dr. Luca Benini and Dr. Patrick Vuillod in formalizing generalized matching and in applying it to library binding. The Author acknowledges support from NSF, under grant MIP-9421129.

References

1. A.Aho, R.Sethi and J. Ullman, "Compilers: Principles, Techniques and Tools," Addison-Wesley, Reading, MA , 1986.
2. A.Aho and M.Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of ACM*, Vol 18, No.6, June 1975, pp.333-340.
3. A.Aho and S.Johnson, "Optimal Code Generation for Expression Trees," *Journal of ACM*, Vol 23, No.3, June 1976, pp. 488-501.
4. 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.
5. C. Bolchini, G. Buonanno et al., "A new switching-level approach to multiple-output function synthesis," in *Proceedings of the International Conference on VLSI Design*, pp. 125-129, January 1995.
6. 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.
7. R. Brayton, G. Hachtel, C. McMullen and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*, Kluwer, 1984.
8. F. Brown. *Boolean reasoning.* Kluwer Academic Publishers, 1990.
9. R. Bryant, "Graph-Based Algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, Vol. C-35, No. 8, August 1986, pp. 677-691.
10. 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.
11. S. Chang, L. Van Ginneken and M. Marek-Sadowska, "Fast Boolean optimization by rewiring," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 262–269, Nov. 1996.
12. 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.
13. 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.
14. K. Cheng and L. Entrena, "Multi-level logic optimization by redundancy addition and removal," in *European Conference on Design Automation*, pp. 373–377, Feb. 1993.
15. E. M. Clarke, K. L. McMillan, X.Zhao, M. Fujita and J. Yang, "Spectral transforms for large Boolean functions with application to technology mapping," in *DAC, Proceedings of the Design Automation Conference*, pp. 54–60, June 1993.

16. J. Cong and Y. Ding, "An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *ICCAD, Proceedings of the International Conference on Computer Aided Design*, 1992, pp. 48-53.
17. J. Darringer, W. Joyner, L.Berman and L. Trevillyan, "LSS: Logic synthesis through local transformations," *IBM Journal of Research and Development*, Vol 25, No 4, pp. 272-280, July 1981.
18. G. De Micheli. *Synthesis and optimization of digital circuits.* McGraw-Hill, 1994.
19. E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni and A. Wang, "Technology mapping in MIS," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 116–119, Nov. 1987.
20. C. Edwards, "Applications of Rademacher-Walsh transform to Boolean function classification and threshold logic synthesis", *IEEE Transactions on Computers*, pp. 48-62, January 1975.
21. M. Garey and D. Johnson, *Computers and intractability*, W.Freeman, New York, 1979.
22. D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel, "Socrates: A System for Automatically synthesizing and optimizing combinational logic," *DAC, Proceedings of the Design Automation Conference*, pp. 79-85, 1986.
23. G.Hachtel, R.Jacobi, K.Keutzer and C.Morrison, "On Properties of Algebraic Transformations and the Synthesis of Multi-fault Irredundant Circuits," *IEEE Transactions on CAD/ICAS* , Vol. 11, No. 3, March 1992, pp.313-321.
24. S. Hurst, D. Miller and J. Muzio, *Spectral techniques in digital logic*, Academic Press, London, United Kingdom, 1985.
25. K. Keutzer, "DAGON: technology binding and local optimization by DAG matching," in *DAC, Proceedings of the Design Automation Conference*, pp. 341–347, June 1987.
26. Y. Kukimoto, R. Brayton, P.Sawkar. "Delay Optimal Technology Mapping by DAG Covering," *DAC, Proceedings of the Design Automation Conference*, 1998, pp. 348-351.
27. W. Kunz and P. Menon, "Multi-level logic optimization by implication analysis," in *Proceeding of the International Conference on Computer-Aided Design*, pp. 6–13, Nov. 1994.
28. E. Lehman, Y. Watanabe, J. Grodstein and H. Harkness, "Logic Decomposition During Technology Mapping," *IEEE Transactions on CAD/ICAS* , Vol. 16, No. 8, August 1997, pp. 813-834.
29. S. Krishnamoorthy and F. Mailhot, "Boolean matching of sequential elements", *DAC, Proceedings of the Design Automation Conference*, pp.691-697, 1994.
30. Y. T. Lai, S. Sastry and M. Pedram, "Boolean matching using binary decision diagrams with applications to logic synthesis and verification," in *ICCD, Proceedings of the International Conference on Computer Design*, pp. 452–458, Oct. 1992.
31. J. Lou, A. Salek and M. Pedram, "An Exact Solution to Simultaneous Technology Mapping and Linear Placement Problem," *ICCAD, Proceedings of the International Conference on Computer Aided Design* , 1997, pp. 671-675.
32. F. Mailhot and G. De Micheli, "Algorithms for technology mapping based on binary decision diagrams and on Boolean operations," *IEEE Transactions on CAD/ICAS*, Vol. 12, No. 5, May 1993, pp.599-620.

33. R. Marculescu, D. Marculescu and M. Pedram, "Logic level power estimation considering spatiotemporal correlations," in *Proceedings of the International Conference on Computer Aided Design*, pp. 294–299, 1994.
34. J. Mohnke and S. Malik, "Permutation and phase independent Boolean comparison," *Integration, The VLSI Journal*, pp. 109–129, Dec. 1993.
35. C. R. Morrison, R. M. Jacoby, and G. D. Hachtel, "Techmap: technology mapping with delay and area optimization", in G. Saucier and P. M. McLellan, editors, *Logic and Architecture Synthesis for Silicon Compilers*, pp. 53–64. North-Holland, Amsterdam, The Netherlands, 1989.
36. J. Rabaey and M. Pedram (Editors), *Low-Power Design Methodologies*, Kluwer Academic Publishers, Boston, MA , 1996.
37. B. Rohfleisch, B. Wurth and K. Antreich, "Logic clause analysis for delay optimization," in *DAC, Proceedings of the Design Automation Conference*, pp. 668–672, June 1995.
38. R. Rudell, *Logic Synthesis for VLSI Design*, Memorandum UCB/ERL M89/49, PhD thesis, U. C. Berkeley, April 1989.
39. H. Savoj, M. J. Silva, R. Brayton and A. Sangiovanni, "Boolean matching in logic synthesis," in *EURO-DAC, Proceedings of the European Design Automation Conference,* pp. 168–174, Sep. 1992.
40. U. Schlichtmann, F. Brglez and M.Herrmann, "Characterization of Boolean functions for rapid matching in EPGA technology mapping," in *DAC, Proceedings of the Design Automation Conference,* pp. 374-379, June 1992.
41. U. Schlichtmann, F. Brglez and P. Schneider, "Efficient Boolean matching based on unique variable ordering," in *International Workshop on Logic Synthesis*, May 1993.
42. F. Somenzi and R. K. Brayton, "Minimization of Boolean relations," in *IEEE, Proceedings of the International Symposium on Circuits and Systems*, pp. 738–473, May 1989.
43. C.-C. Tsai and M. Marek-Sadowska, "Boolean matching using generalized Reed-Muller forms," in *DAC, Proceedings of the Design Automation Conference*, pp. 339–344, June 1994.
44. P. Vuillod, L. Benini and G. De Micheli, "Re-mapping for Low Power under Timing Constraints," *ISLPED, IEEE Symposium on Low Power Electronics and Design*, 1997, pp. 287-292.
45. P. Vuillod, L. Benini, G. De Micheli, "Generalized Matching from Theory to Application," *ICCAD, Proceedings of the International Conference on Computer Aided Design* , pp. 13-20, 1997.
46. K.-H. Wang and T.-T. Hwang, "Boolean matching for incompletely specified Functions," in *DAC, Proceedings of the Design Automation Conference*, pp. 48-53, June 1995.
47. K.-H. Wang, T.-T. Hwang and C.Chen, "Exploiting communication complexity in Boolean matching," *IEEE Transactions on CAD/ICAS* Vol. 15, N0. 10, pp. 1249-1256, October 1996. pp. 48-53, June 1995.
48. Y. Watanabe, L. M. Guerra and R. K. Brayton, "Permissible functions for multioutput components in combinational logic optimization," *IEEE transactions on CAD/ICAS* Vol. 15, no. 7, pp. 734–744, July 1996.
49. J. Yang and G. De Micheli, "Spectral techniques for technology mapping," *CSL Report,* CSL-TR-91-498, 1991.