Boolean Matching Based on Boolean Unification

Kuang-Chien Chen
Fujitsu America Inc.
3055 Orchard Drive
San Jose, CA, 95134
U.S.A.

May 21, 1993

Abstract

We consider the problem of detecting the equivalence of two single-output Boolean functions, considering the permutation and complementation of their inputs, complementation of outputs, and their associated don't-care sets. This is often referred to as the **Boolean matching** problem. Boolean matching is a verification problem, and it has important applications in logic synthesis problems such as technology-mapping [1, 2, 3]. In this paper, we present a new algorithm for solving the Boolean matching problem which is based on Boolean unification and branch-and-bound techniques. We have applied this algorithm to the task of technology-mapping for cell-based designs, and experimental results show that it is an efficient and effective algorithm. Comparisons with existing Boolean matching algorithms will also be presented.

1 Introduction and Review of Previous Work

In design verification, we often need to test the equivalence of two combinational circuits. If the correspondence of the inputs of two circuits are given, their equivalence can be tested by algorithms based on OBDDs (ordered binary decision diagrams [4]). However, if the correspondence between the inputs of two circuits are not given and their signal polarities can be inverted, then this is a difficult problem which is often referred to as **Boolean matching.** In logic synthesis problems such as technology-mapping, we also need to match a function in a multi-level circuit with functions of the cells in a given technology library. An algorithm for technology-mapping based on the matching of functions represented as tree-patterns was proposed in Dagon [5] and it has been used in many technology mappers. This algorithm, however, only considered graph isomorphism in the matching process, and requires the enumeration of all possible patterns which is not feasible in some situations. Therefore, two functions which are Boolean equivalent but not structurally equivalent may not be detected by this algorithm.

The use of Boolean matching in technology-mapping was proposed in [1]. This Boolean matching algorithm basically tries all possible ways in matching two functions by considering input negation and permutation, and output negation (so call NPN properties [8]). To speedup processing, *symmetry* and *unateness* of input variables are utilized. *Don't-cares* associated with the two functions under matching are also considered by a lattice-based method. However, the lattice-based method is efficient only for a small number of inputs. So the use of don't-cares is limited to functions of 4 inputs.

Another algorithm for Boolean matching was proposed by Savoj, *et al.* [2]. This algorithm used a branch-and-bound method to match the variables in a top-down fashion. Symmetry of the variables and bounding conditions based on minterm count are used to trim the search space. This is one of the most general algorithm that considers the use of don't-cares in Boolean matching.

Recently, yet another algorithm for Boolean matching was developed which is based on computing semi-canonical forms for functions and OBDD techniques [3]. This algorithm, although very efficient, does not consider don't-cares in the matching process.

In this work, we present a new algorithm for Boolean matching which can handle don't-cares and properties such as NPN equivalence, constant inputs, and bridging of inputs. It is based on Boolean unification and branch-and-bound techniques. We have applied this algorithm to the technology-mapping for cell-based designs, and experimental results show that this is not only a general but also very efficient algorithm for Boolean matching using don't-cares.

2 Terminology and Notation

Let $f(x_1, x_2, \cdots, x_m)$ and $g(y_1, y_2, \cdots, y_n)$ be two Boolean functions with associated don't-care sets $f_d(x_1, x_2, \cdots, x_m)$ and $g_d(y_1, y_2, \cdots, y_n)$, respectively. The Boolean matching problem is to determine whether f is equivalent to g, denoted by $f \cong g$, by considering their don't-care sets and allowing permutation and complementation of the inputs and complementation of outputs.

The **cofactor** of a function f with respect to a literal x_i (or $\bar{x}_i$), denoted by f_{x_i} (or $f_{\bar{x}_i}$), is a new function obtained by substituting 1 (or 0) for x_i (or $\bar{x}_i$) in f. The **Shannon expansion** of f with respect to a variable x_i is

$$f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i} \quad .$$

The **consensus operator** applied to a function f with respect to a variable x_i is denoted by $\mathcal{C}_{x_i} f$ and defined as:

$$\mathcal{C}_{x_i} f = f_{x_i} f_{\bar{x}_i} \quad .$$

The **smoothing operator** applied to a function f with respect to a variable x_i is denoted by $\mathcal{S}_{x_i} f$

and defined as:

$$\mathcal{S}_{x_i} f = f_{x_i} + f_{\bar{x}_i} \quad .$$

Both $\mathcal{C}_{x_i} f$ and $\mathcal{S}_{x_i} f$ are functions independent of x_i.

3 Boolean Unification Algorithms

Boolean unification (i.e., solution of Boolean equations), as discussed in [6, 7], is a basic inference mechanism in algebraic manipulation of formulas. Given a formula, Boolean unification derives a general solution (called **the most general unifier**, abbreviated as **mgu** henceforth) which contains some newly introduced variables to represent all the possible solutions (if there exist any solution). Then, particular solutions can be obtained by substituting arbitrary functions to the new variables in the general solution. Boolean unification has been applied to solve problems in test pattern generation, logic minimization, design rectification and verification, and Boolean relation minimization [7, 10].

Given a relation $R(p_1, p_2, \cdots, p_m, q_1, q_2, \cdots, q_n) = 0$ where $q_1, q_2, \cdots, q_n$ are *independent variables*, the Boolean unification problem is to find the most general solutions for $p_1, p_2, \cdots, p_m$ which satisfy the given relation. These general solutions are usually functions of $q_1, q_2, \cdots, q_n$ and some newly introduced variables $r_1, r_2, \cdots, r_m$:

$$\begin{aligned}
p_1 &= mgu_1(q_1, q_2, \cdots, q_n, r_1, r_2, \cdots, r_m) \\
p_2 &= mgu_2(q_1, q_2, \cdots, q_n, r_1, r_2, \cdots, r_m) \\
&\cdots \\
p_m &= mgu_m(q_1, q_2, \cdots, q_n, r_1, r_2, \cdots, r_m)
\end{aligned}$$

Variables $r_1, r_2, \cdots, r_m$ can be *arbitrary* functions of $q_1, q_2, \cdots, q_n$ (including constants). Boolean unification can be regarded as a method for translating a relational expression (i.e., $R(p_1,\ p_2,\ \cdots,\ p_m,\ q_1,\ q_2,\ \cdots,\ q_n) = 0$) into functional expressions (i.e., $p_1 = mgu_1, \cdots, p_m = mgu_m$) by introducing new variables.

Example 1 *Suppose the following relation is given:*

$$f(p_1, p_2, q_1, q_2) = \bar{p}_1 q_1 q_2 + \bar{p}_1 \bar{q}_1 \bar{q}_2 + \bar{p}_2 q_1 q_2 + \bar{p}_2 \bar{q}_1 \bar{q}_2 + p_1 p_2 q_1 \bar{q}_2 + p_1 p_2 \bar{q}_1 q_2 = \phi \tag{1}$$

Most general unifiers for p_1 and p_2 are (an algorithm for obtaining these mgu's is discussed later):

$$\begin{aligned}
p_1 &= q_1 q_2 + \bar{q}_1 \bar{q}_2 + r_1 \bar{r}_2 \\
p_2 &= q_1 q_2 + \bar{q}_1 \bar{q}_2 + r_2
\end{aligned}$$

Using these mgu's, relation (1) is always satisfied no matter what are the functions of q_1 and q_2. By substituting any functions (or constants) into r_1 and r_2, particular solutions can be obtained. For example, let $r_1 = q_1$ and $r_2 = q_2$, we have:

$$p_1 = q_1 + \bar{q}_2$$
$$p_2 = \bar{q}_1 + q_2.$$

Now we discuss a Boolean unification algorithm developed by Boole [6, 7] which is the basis for our Boolean matching algorithm. Boole's algorithm is based on recursive Shannon expansion of a given formula, and it derives the most general unifiers in a *bottom-up* fashion. Shown in Figure 1 is Boole's unification algorithm (it is modified from the original algorithm for ease of understanding). The following conditions are checked at each step of the expansion process:

(1) If $G_{p_k} = L_{p_k} R_{p_k} = \phi$ and $k = m$ (lines 4-5): no more p-variables need to be unified. $mgu[p_k]$ is calculated as $(L_{p_k} \oplus R_{p_k} \oplus 1) r_k \oplus L_{p_k}$ (line 6).

(2) If $G_{p_k} = L_{p_k} R_{p_k} = \phi$ and $k < m$ (lines 4, 8): the remaining variables $p_{k+1}, \cdots, p_m$ can be any function. Therefore, the mgu's for $p_{k+1}, \cdots, p_m$ are simply new variables which can be any function (lines 9-11; notice that we can also use the original variables without introducing these new variables). After substituting $p_{k+1}, \cdots, p_m$ by their mgu's in L_{p_k} and R_{p_k} (lines 12-15), the $mgu[p_k]$ is calculated (line 16).

(3) If $G_{p_k} = L_{p_k} R_{p_k} \neq \phi$ but no more p-variables to be unified (line 18): we can not derive mgu's and unification fails.

(4) If $G_{p_k} = L_{p_k} R_{p_k} \neq \phi$ and $k < m$: the unification algorithm is applied recursively on G_{p_k} to obtain $mgu[p_{k+1}], \cdots, mgu[p_m]$ (line 19), and then the results are substituted into L_{p_k} and R_{p_k} to obtain $mgu[p_k]$ (lines 20-24).

Notice that Boole's algorithm introduces at most m new variables for unifying $p_1, \cdots, p_m$. Also, different orders of applying the partitioning variables may result in a different set of mgu's (although they still represent all the possible solutions). This is because case (2) may happen sooner or later depending on the order of applying the partitioning variables.

Example 2 *Let us consider the formula*

$$f(p_1, p_2, q_1, q_2) = \bar{p}_1 q_1 q_2 + \bar{p}_1 \bar{q}_1 \bar{q}_2 + \bar{p}_2 q_1 q_2 + \bar{p}_2 \bar{q}_1 \bar{q}_2 + p_1 p_2 q_1 \bar{q}_2 + p_1 p_2 \bar{q}_1 q_2$$

used in Example 1. We first partition using variable p_1 and obtain:

```
/* Initially $R$ is a function of $p_1, p_2, \cdots, p_m, q_1, q_2, \cdots, q_n$.
   Boolean unification is done by calling BU_Boole($R$, $mgu$, 1).
*/
BU_Boole($R$, $mgu$, $k$) {
        $L_{p_k}$ = cofactor($R$, $\bar{p}_k$);
        $R_{p_k}$ = cofactor($R$, $p_k$);
        $G_{p_k}$ = $\mathcal{C}_{p_k} R$ = $L_{p_k} R_{p_k}$;
        if($G_{p_k}$ = $\phi$) {
           /* case (1): no more $p$-variable to unify */
           if($k = m$) {
                 $mgu[p_k]$ = $(L_{p_k} \oplus R_{p_k} \oplus 1) r_k \oplus L_{p_k}$;
                 return;
           }
           /* case (2): $p_{k+1}, \cdots, p_m$ not unified yet */
           if($k < m$) {
                 /* $p_{k+1}, \cdots, p_m$ can be any function */
                 for(i = k+1; i<=m; i++) {
                     $mgu[p_i]$ = $r_i$;
                 }
                 /* substitute the mgu's obtained so far into $L_{p_k}$ and $R_{p_k}$ */
                 for(i = k+1; i<=m; i++) {
                     $L_{p_k}$ = $\mathcal{S}_{p_i}(L_{p_k}(p_i \overline{\oplus} mgu[p_i]))$;
                     $R_{p_k}$ = $\mathcal{S}_{p_i}(R_{p_k}(p_i \overline{\oplus} mgu[p_i]))$;
                 }
                 $mgu[p_k]$ = $(L_{p_k} \oplus R_{p_k} \oplus 1) r_k \oplus L_{p_k}$;
                 return;
           }
        }
        /* case (3): $G_{p_k} \neq \phi$ but no more $p$-variable to unify. */
        if($k = m$) report_failure;

        /* case (4): obtain the mgu's for $p_{k+1}, \cdots, p_m$ recursively */
        BU_Boole($G_{p_k}$, $mgu$, $k+1$);
        /* substitute the mgu's obtained so far into $L_{p_k}$ and $R_{p_k}$ */
        for(i = k+1; i<=m; i++) {
             $L_{p_k}$ = $\mathcal{S}_{p_i}(L_{p_k}(p_i \overline{\oplus} mgu[p_i]))$;
             $R_{p_k}$ = $\mathcal{S}_{p_i}(R_{p_k}(p_i \overline{\oplus} mgu[p_i]))$;
        }
        $mgu[p_k]$ = $(L_{p_k} \oplus R_{p_k} \oplus 1) r_k \oplus L_{p_k}$;
        return;
}
```

Figure 1: Boole's Boolean unification algorithm.

$$
\begin{aligned}
L_{p_1} &= f_{\bar{p}_1} = q_1q_2 + \bar{q}_1\bar{q}_2 + \bar{p}_2q_1q_2 + \bar{p}_2\bar{q}_1\bar{q}_2 \\
R_{p_1} &= f_{p_1} = \bar{p}_2q_1q_2 + \bar{p}_2\bar{q}_1\bar{q}_2 + p_2q_1\bar{q}_2 + p_2\bar{q}_1q_2 \\
G_{p_1} &= L_{p_1}R_{p_1} = \bar{p}_2q_1q_2 + \bar{p}_2\bar{q}_1\bar{q}_2
\end{aligned}
$$

Since $G_{p_1} \neq \phi$, the unification procedure is applied recursively on G_{p_1}. Now p_2 is the partitioning variable. We have:

$$
\begin{aligned}
L_{p_2} &= q_1q_2 + \bar{q}_1\bar{q}_2 \\
R_{p_2} &= \phi \\
G_{p_2} &= L_{p_2}R_{p_2} = \phi
\end{aligned}
$$

Since $G_{p_2} = \phi$, the mgu for p_2 is calculated as:

$$mgu[p_2] = (L_{p_2} \oplus R_{p_2} \oplus 1)r_2 \oplus L_{p_2} = q_1q_2 + \bar{q}_1\bar{q}_2 + r_2$$

Substituting p_2 by $mgu[p_2]$ into L_{p_1} and R_{p_1}, we have:

$$
\begin{aligned}
L_{p_1} &= q_1q_2 + \bar{q}_1\bar{q}_2 \\
R_{p_1} &= q_1\bar{q}_2r_2 + \bar{q}_1q_2r_2
\end{aligned}
$$

Then, mgu for p_1 is calculated as:

$$mgu[p_1] = (L_{p_1} \oplus R_{p_1} \oplus 1)r_1 \oplus L_{p_1} = q_1q_2 + \bar{q}_1\bar{q}_2 + r_1\bar{r}_2$$

We can see that to perform Boolean unification efficiently, it is desirable to reach the terminal condition $G_{p_k} = L_{p_k}R_{p_k} = \phi$ as soon as possible. But as we will discuss later, this is not desirable when Boolean unification is applied to Boolean matching.

4 Boolean Matching Using Boolean Unification

The basic idea of Boolean matching using Boolean unification is as follows: given $f(x_1, x_2, \cdots, x_m)$, $g(y_1, y_2, \cdots, y_n)$ and their don't-care sets $f_d(x_1, x_2, \cdots, x_m)$ and $g_d(y_1, y_2, \cdots, y_n)$, respectively, we can treat the y-variables as independent variables, and then apply Boolean unification to solve the relation $fg + \bar{f}\bar{g} + f_d + g_d = 1$, or equivalently $\bar{f}g\bar{f}_d\bar{g}_d + f\bar{g}\bar{f}_d\bar{g}_d = 0$. If unification succeeds, then a general solution of the following form can be obtained:

$$
\begin{aligned}
x_1 &= mgu_1(y_1, y_2, \cdots, y_n, r_1, r_2, \cdots, r_m) \\
x_2 &= mgu_2(y_1, y_2, \cdots, y_n, r_1, r_2, \cdots, r_m) \\
&\cdots \\
x_m &= mgu_m(y_1, y_2, \cdots, y_n, r_1, r_2, \cdots, r_m)
\end{aligned}
$$

where $r_1, r_2, \cdots, r_m$ are new variables that can be arbitrary functions of the y-variables. Then, we can assign appropriate values or functions to the new variables so that a match between x- and y-variables is obtained. If no mgu's can be obtained by Boolean unification, then $f \ncong g$.

However, even though we can obtain the mgu's by Boolean unification, it is very difficult to determine the functions of the new variables $r_1, r_2, \cdots, r_m$ simultaneously to find a match. Instead, we found that applying unification procedure in a branch-and-bound process is an effective way to solve the problem. Whether a variable x_i can be unified and matched to a variable y_j (or $\bar{y}_j$) can be determined efficiently as will be discussed shortly.

Shown in Figure 2 is the algorithm **BM_BU** for Boolean matching using Boolean unification. It finds a match *implicitly* using unification techniques but without actually deriving the mgu's and functions for the new variables. Given functions f and g, and their don't-care sets f_d and g_d respectively, BM_BU first derives the relation $R = \bar{f}g\bar{f}_d\bar{g}_d + f\bar{g}\bar{f}_d\bar{g}_d$ (line 1). Then, $y_1, y_2, \cdots, y_n$ are treated as independent variables and Boolean unification is applied in a bottom-up branch-and-bound fashion to find the match between x- and y- variables by Procedure BU_branch_and_bound (line 10). To avoid calculating cofactors repeatedly in the branch-and-bound process, the cofactors are pre-calculated (lines 3-8) based on a good ordering of the partitioning variables (line 2; we will discuss this further later).

In Procedure BU_branch_and_bound (Figure 3), we try to find a match for variable x_i. Since L_{x_i} and R_{x_i} contains variables $x_{i+1}, \cdots, x_m$, we need to substitute the matches found so far for these variables (lines 3-7). Then, we examining each unmatched variable y_j to see whether it can match x_i (if *input bridging* is allowed, then line 9 should be deleted). Recall from previous discussions that the mgu for x_i is:

$$x_i = (L_{x_i} \oplus R_{x_i} \oplus 1)r_i \oplus L_{x_i}.$$

This equation can be simplified to:

$$x_i = L_{x_i}\bar{r}_i + \overline{R_{x_i}}r_i.$$

To match x_i with y_j, we must have:

$$L_{x_i}\bar{r}_i + \overline{R_{x_i}}r_i \subseteq y_j \quad \text{and} \quad y_j \subseteq L_{x_i}\bar{r}_i + \overline{R_{x_i}}r_i,$$

or equivalently:

$$L_{x_i}\bar{y}_j\bar{r}_i = \phi, \;\; \overline{R_{x_i}}\bar{y}_j r_i = \phi, \quad \overline{L_{x_i}}R_{x_i}y_j = \phi, \quad \overline{L_{x_i}}y_j\bar{r}_i = \phi, \quad R_{x_i}y_j r_i = \phi.$$

After further simplification, these equations can be translated into the following two necessary and sufficient conditions which determine whether y_j can match x_i (line 11):

```
BM_BU(f, f_d, g, g_d) {
    /* f and f_d are functions of {x_1, x_2, ..., x_m}.
       g and g_d are functions of {y_1, y_2, ..., y_n}.
    */
    R = f̄ g f̄_d ḡ_d + f ḡ f̄_d ḡ_d;
    Order {x_1, x_2, ..., x_m} by a branch-and-bound procedure, so that the
    partitioning process can go as deep as possible.  To keep notations simple,
    we still use {x_1, x_2, ..., x_m} to represent the result obtained.

    /* Derive the cofactors for later use */
    G = R;
    for(i = 1; i ≤ m; i++) {
        L[i] = cofactor(G, x̄_i);
        R[i] = cofactor(G, x_i);
        G = L[i]R[i];
    }
    /* G should be φ after partitioning using all the x-variables.  */
    if(G ≠ φ) report_failure;

    /* solve the matching problem bottom-up starting from x_m */
    if(BU_branch_and_bound(L, R, m, match))
       return match;
}
```

Figure 2: A Boolean matching algorithm based on Boolean unification

$$\overline{L_{x_i}} R_{x_i} y_j = \phi \;\text{ and }\; L_{x_i} \overline{R_{x_i}} \bar{y}_j = \phi.$$

These two equations are used as bounding conditions to abort unsuccessful matches. After a match is found, we call the branch-and-bound procedure recursively to see whether this is a good choice (lines 13-17). If y_j does not match, we also try to match with $\bar{y}_j$ (lines 19-26).

There are several important issues:

- The existence of mgu's in Algorithm BM_BU does not always mean $f \cong g$. It is only a necessary condition. However, if we extend the definition of Boolean matching to: $f \cong g$ if there exist functions $\psi_1, \psi_2, \cdots, \psi_m$ such that $fg + \bar{f}\bar{g} + f_d + g_d = 1$ when $x_i = \psi_i(y_1, \cdots, y_n), i = 1, \cdots, m$, then being able to derive mgu's is a necessary and sufficient condition for $f \cong g$.
- We do not consider input bridging (including dual-rail inputs) and constant inputs in Figure 3, but these cases can be easily taken into account in Procedure BU_branch_and_bound.
- If don't-cares are not present, we can usually take advantage of the **symmetry** that exists among the input variables of f or g. In applications such as technology-mapping, the function of a cell in the library (let's assume it is the g function) usually does not have don't-cares. In this case, symmetry of the variables can be easily utilized as shown in Figure 3 (line 10),
- The Boolean matching algorithm proposed in [2] also used a branch-and-bound method, but it is performed in a top-down fashion and minterm counts of the cofactors were used as the bounding conditions. As we can see later from the experimental results, the bounding conditions in Algorithm BM_BU are more effective and result in a much more efficient algorithm.

As discussed in Algorithm BU_Boole, if $G_{x_i} = L_{x_i} R_{x_i} = \phi$ happens, variables $x_{i+1}, \cdots, x_m$ can potentially match any y-variable and we may have to spend much time on the branch-and-bound process. To reduce the search space, it is desirable to reach the condition $G_{x_i} = 0$ *as late as possible*. Therefore, in the beginning of Algorithm BM_BU, we have used another branch-and-bound procedure to find an ordering of the partitioning variables so that we can go as deep as possible in the partitioning process (the details of this procedure is not shown in Figure 2 for clarity). Experiments show that tremendous speedup is achieved by using this technique.

5 Application in Technology-Mapping and Experimental Results

We applied the Boolean matching algorithm discussed in previous sections to the technology-mapping of cell-based designs. To make comparisons with the Boolean matching algorithm in

```
BU_branch_and_bound($\mathcal{L}$, $\mathcal{R}$, $i$, $match$) {
    $L_{x_i}$ = $\mathcal{L}[i]$;
    $R_{x_i}$ = $\mathcal{R}[i]$;
    /* substitute previous matches, $m$ is the number of $x$-variables */
    for($k = i + 1$; $k \leq m$; $k$++) {
        $y_t$ = $match[x_k]$;
        $L_{x_i}$ = $\mathcal{S}_{x_k}(L_{x_i}(x_k \overline{\oplus} y_t))$;
        $R_{x_i}$ = $\mathcal{S}_{x_k}(R_{x_i}(x_k \overline{\oplus} y_t))$;
    }
    /* choose a match by branch-and-bound */
    for($j = 1$; $j \leq n$; $j$++) {
        /* skip if $y_j$ is already matched to other $x$-variable */
        if(already_used_in_match($match$, $y_j$))
            continue;
        /* symmetry should be used only when $g_d = \phi$ */
        if(($y_k$ is symmetric to $y_j$) and ($y_k$ and $\bar{y}_k$ fail to match $x_i$))
            continue;
        /* test whether $y_j$ is a candidate */
        if($\bar{L}_{x_i} R_{x_i} y_j = \phi$ and $L_{x_i} \bar{R}_{x_i} \bar{y}_j = \phi$) {
            $match[x_i]$ = $y_j$;
            /* call recursively to see whether this is a good choice */
            if(BU_branch_and_bound($\mathcal{L}$, $\mathcal{R}$, $i - 1$, $match$) {
                return $true$;
            } else {
                /* undo the choice */
                $match[x_i]$ = $null$;
            }
        }
        /* test whether $\bar{y}_j$ is a candidate */
        if($\bar{L}_{x_i} R_{x_i} \bar{y}_j = \phi$ and $L_{x_i} \bar{R}_{x_i} y_j = \phi$) {
            $match[x_i]$ = $\bar{y}_j$;
            /* call recursively to see whether this is a good choice */
            if(BU_branch_and_bound($\mathcal{L}$, $\mathcal{R}$, $i - 1$, $match$) {
                return $true$;
            } else {
                /* undo the choice */
                $match[x_i]$ = $null$;
        }
    }
    /* have tried all possibilities */
    return $false$;
}
```

Figure 3: The bottom-up branch-and-bound procedure used in BM_BU.

SIS [2] easier, we integrated our Boolean matching algorithm into SIS and utilized its existing procedures for technology mapping. We concentrated on comparing the performance between these two Boolean matching algorithms.

A network is first decomposed into trees, and then Boolean matching is used to match the functions in each tree with functions in the cell-library. Don't-cares are derived for functions in the trees, but usually no don't-cares are specified for functions in the cell-library. We used an Actel library for the experiment, and the results are shown in Table 1. As we can see, Algorithm BM_BU is much faster than the Boolean matching algorithm in SIS and uses less memory. This is mainly due to the effective bounding conditions we used for unification in the branch-and-bound process, and also a good order of applying the partitioning variables. The small differences in the network area result from different input phases used when mapping to cells such as XOR gate.

6 Conclusions and Future Work

We presented a Boolean matching algorithm based on Boolean unification and branch-and-bound techniques. This algorithm considers don't-cares and can handle various matching constraints. Preliminary experimental results show that this algorithm is much faster than existing Boolean matching algorithms that consider don't-cares. Future work will be on using *filters* [2, 3] to further speed up the matching process and use the Boolean matching algorithm as a core to develop more powerful logic synthesis algorithms.

7 Acknowledgements

The author would like to thank Prof. Brayton, Dr. Savoj and Mr. Silva for providing the Boolean matching program in SIS for the experiment.

Table 1: Comparison of Boolean matching algorithms for technology mapping.

Function			SIS		BM_BU	
Name	#In	# Out	Area	Time	Area	Time
9symml	9	1	96	28.1	100	25.9
alu2	10	6	204	629.1	203	65.3
alu4	14	8	out of	memory	360	114.6
apex6	135	99	436	366.6	401	156.2
apex7	49	37	out of	memory	136	183.2
cc	21	20	29	1305.4	26	20.4
cht	47	36	63	38.2	67	9.3
comp	32	3	58	17.4	60	11.0
cordic	23	2	41	8.3	40	6.9
dalu	75	16	out of	memory	822	406.7
des	256	245	1989	832.1	2003	541.8
example2	85	66	150	264.7	149	38.8
f51m	8	8	58	334.1	63	49.8
i1	25	16	22	47.3	22	2.9
i2	201	1	83	31.0	83	25.1
i3	132	6	50	18.7	50	18.1
i4	192	6	130	32.6	130	30.4
i5	133	66	187	36.4	192	33.6
i6	138	67	267	933.4	250	140.8
i7	199	67	out of	memory	344	557.6
i8	133	81	1458	443.0	1377	564.5
i9	88	63	out of	memory	435	183.7
k2	45	45	out of	memory	1090	194.1
lal	26	19	78	13.4	78	11.9
my_adder	33	17	48	3.1	48	2.3
pair	173	137	731	540.3	752	107.2
parity	16	1	15	0.3	15	0.1
pcler8	27	17	32	198.7	33	9.2
pm1	16	13	32	4.9	36	4.2
rot	135	107	324	91.1	318	68.6
term1	34	10	215	66.3	213	55.9
too_large	38	3	446	99.1	436	85.5
ttt2	24	21	124	181.3	133	27.1
vda	17	39	518	237.3	520	92.0
z4ml	7	4	31	62.6	32	19.9

References

[1] F. Mailhot and G. D. Micheli, "Technology Mapping Using Boolean Matching," *Proceedings of EDAC*, Mar. 1990.

[2] H. Savoj and M. J. Silva, "Boolean Matching in Logic Synthesis," *Proceedings of Euro-DAC*, 1992.

[3] J. R. Burch and D. E. Long, "Efficient Boolean Function Matching," *Proceedings of ICCAD*, 1992.

[4] R. E. Bryant, "Graph-Base Algorithms for Boolean Function Manipulation," *IEEE TCAD*, Vol. C-35, No. 8, Aug. 1986.

[5] K. Keutzer, "Dagon: Technology Binding and Local Optimization by DAG Matching," *Proceedings of DAC*, 1987.

[6] U. Martin and T. Nipkow, "Unification in Boolean Ring", *Journal of Automated Reasoning*, Vol. 4, pp. 381-396, 1988.

[7] U. Martin and T. Nipkow, "Boolean Unification - The Story So Far", *Journal of Symbolic Computation*, Vol. 7, pp.275-293, 1989.

[8] S. Muroga, "Threshold Logic and its Applications," *John Wiley*, 1971.

[9] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE TCAD*, Vol. 11, No. 1, Jan. 1992.

[10] M. Fujita, Y. Matsunaga, T. Kakuda, and K-C Chen, "On Application of Boolean Unification to Combinational Logic Synthesis," *Proceedings of ICCAD*, 1991.