Simulation and SAT-Based Boolean Matching for Large Boolean Networks [*]

Kuo-Hua Wang, Chung-Ming Chan, and Jung-Chang Liu
Dept. of Computer Science and Information Engineering
Fu Jen Catholic University
Hsinchuang City, Taipei County 24205, Taiwan, R.O.C.
khwang@csie.fju.edu.tw

ABSTRACT

Boolean matching is to check the equivalence of two target functions under input permutation and input/output phase assignment. This paper addresses the permutation independent (P-equivalent) Boolean matching problem. We will propose a matching algorithm seamlessly integrating Simulation and Boolean Satisfiability (S&S) techniques. Our proposed algorithm will first utilize functional properties like unateness and symmetry to reduce the searching space. In the followed simulation phase, three types of input vector generation and checking method will be used to match the inputs of two target functions. Experimental results on large benchmarking circuits demonstrate that our matching algorithm is indeed very effective and efficient to solve Boolean matching for large Boolean networks.

Categories and Subject Descriptors: B.6.3 [**Hardware**] Logic Design: Design Aids - Verification.
General Terms: Algorithms, Design, Verification.
Keywords: Boolean Matching, Simulation and SAT.

1. INTRODUCTION

Logic simulation technique had been widely used in design verification and debugging over a long period of time. The major disadvantage of using simulation comes from the fact that it is very time consuming and almost impossible to catch completely the functionality of a very large Boolean network. To solve this issue, Boolean satisfiability (SAT) technique was proposed and exploited in many industrial formal verification tools. In recent years, the technique of combining simulation and SAT was popular and successfully applied in many verification and synthesis problems like equivalence checking [1] and logic minimization [2]-[4].

Boolean matching is to check whether two functions are equivalent or not under input permutation and input/output phase assignment (so-called *NPN-class*). The important applications of Boolean matching involve the verification of two circuits under unknown input correspondences, cell-library binding, and table look-up based FPGA's technology mapping. In the past decades, various Boolean matching techniques had been proposed and some of these approaches were discussed in the survey paper [5]. Among those previously proposed approaches, computing *signatures* [5][6] and transforming into *canonical form* [7][8] of Boolean functions were the most successful techniques to solve Boolean matching. Recently, SAT technique was also applied for Boolean matching with don't cares [9]. Most of these techniques were proposed to handle completely specified functions, comparatively little research had focused on dealing with Boolean functions with don't cares [6][9].

The first and foremost issue for Boolean matching is the data structure for representing Boolean functions. As we know that many prior techniques used truth table, sum of products (SOPs), and Binary Decision Diagrams (BDDs) to represent the target functions during the matching process, they suffer from the same memory explosion problem. By our knowledge, many types of Boolean functions can not be represented by SOPs for large input set and the memory space will explode while constructing their BDD's. Therefore, these Boolean matching techniques were constrained to apply for small to moderate Boolean networks (functions). To address the above issues, And-Inverter Graphs (AIGs) [10] had been utilized and successfully applied in verification and synthesis problems [4][10]. In this paper, we will propose a permutation independent (P-equivalent) Boolean matching algorithm for large Boolean functions.

This paper is organized as follows. Section 2 gives a brief research background on our work. Section 3 shows our procedure of detecting functional properties. Some definitions and notations are given in Section 4. Section 5 presents our simulation strategy for distinguishing the input variables of Boolean functions. Section 6 and Section 7 show our S&S-based matching algorithm with implementation issues and the experimental results, respectively. Section 8 concludes this paper and suggests some directions for the future work.

2. BACKGROUND

2.1 Boolean Matching

Boolean matching is to check the equivalence of two target functions $f(X)$ and $g(Y)$ under input permutation and input/output phase assignment. To solve this problem, we have to search a *feasible* mapping ψ such that $f(\psi(X)) = g(Y)$ (or $\bar{g}(Y)$). It is impractical to search all possible mappings because the time complexity is $O(n! \cdot 2^{n+1})$, where n is the number of input variables. Among those previously proposed techniques for Boolean matching, *signature*

[*]This work was supported in part by the National Science Council of Taiwan under Grant NSC 96-2221-E-030-014-MY2.

Table 1: Definition and S&S-Based Checking of Functional Properties.

Name	Property		S&S Checking		
	Definition	Notation	Disjoint	Removal_Condition	SAT_Check
Positive Unate	$f_{\bar{x_i}} \subseteq f_{x_i}$	$PU(x_i)$	x_i	$f(v_1) = 1,\ val(v_1, x_i) = 0$	$f_{\bar{x_i}} \cdot \bar{f}_{x_i} = 0$
Negative Unate	$f_{x_i} \subseteq f_{\bar{x_i}}$	$NU(x_i)$	x_i	$f(v_1) = 1,\ val(v_1, x_i) = 1$	$\bar{f}_{\bar{x_i}} \cdot f_{x_i} = 0$
NE Symmetry	$f_{\bar{x_i}x_j} = f_{x_i\bar{x_j}}$	$NE(x_i, x_j)$	x_i, x_j	$val(v_1, x_i) \neq val(v_1, x_j)$	$f_{x_ix_j} \oplus f_{x_i\bar{x_j}} = 0$
E Symmetry	$f_{\bar{x_i}\bar{x_j}} = f_{x_ix_j}$	$E(x_i, x_j)$	x_i, x_j	$val(v_1, x_i) = val(v_1, x_j)$	$f_{\bar{x_i}\bar{x_j}} \oplus f_{x_ix_j} = 0$
Single Variable Symmetry	$f_{\bar{x_i}\bar{x_j}} = f_{x_i\bar{x_j}}$	$SV(x_i, \bar{x_j})$	x_i	$val(v_1, x_j) = 0, \forall x_j \in X - \{x_i\}$	$f_{\bar{x_i}\bar{x_j}} \oplus f_{x_i\bar{x_j}} = 0$
	$f_{\bar{x_i}x_j} = f_{x_ix_j}$	$SV(x_i, x_j)$	x_i	$val(v_1, x_j) = 1, \forall x_j \in X - \{x_i\}$	$f_{\bar{x_i}x_j} \oplus f_{x_ix_j} = 0$
	$f_{\bar{x_i}\bar{x_j}} = f_{\bar{x_i}x_j}$	$SV(x_j, \bar{x_i})$	x_j	$val(v_1, x_i) = 0, \forall x_i \in X - \{x_j\}$	$f_{\bar{x_i}\bar{x_j}} \oplus f_{\bar{x_i}x_j} = 0$
	$f_{x_i\bar{x_j}} = f_{x_ix_j}$	$SV(x_j, x_i)$	x_j	$val(v_1, x_i) = 1, \forall x_i \in X - \{x_j\}$	$f_{x_i\bar{x_j}} \oplus f_{x_ix_j} = 0$

is one of the most effective approaches. Various signatures were defined to characterize input variables of Boolean functions [5]. Since these signatures are invariant under the permutation or complementation of input variables, the input variables with different signatures can be distinguished with each other and many infeasible mappings can be pruned quickly. However, it had been proved that signatures have the inherent limitation to distinguish all input variables for those functions with $\mathcal{G}$-*symmetry* [11].

2.2 Boolean Satisfiability

The Boolean Satisfiability (SAT) problem is to find a variable assignment to satisfy a given conjunctive normal form (CNF) or prove it is equal to the constant 0. Despite the fact that SAT problem is NP-complete, many advanced techniques like *non-chronological backtracking*, *conflict driven clause learning*, and *watch literals* have been proposed and implemented in the state-of-art SAT solvers [13][14]. Consequently, SAT technique has been successfully applied on solving many EDA problems [15] over the past decades. Among these applications, combinational equivalence checking (CEC) is an important one of utilizing SAT solver to check the equivalence of two combinational circuits. The following briefly describes the concept of SAT-based equivalence checking. Consider two functions (circuits) f and g to be verified. A *miter* circuit with functionality $f \oplus g$ is constructed first and then transformed into a SAT instance (circuit CNF) by simple gate transformation rules. If this circuit CNF can not be satisfied by the SAT solver, then f and g are equivalent; otherwise, they are not equivalent.

2.3 And-Inverter Graph

And-Invert Graphs (AIGs) is a directed acyclic graph which can be used as the structural representation of Boolean functions. It consists of three types of nodes: primary input, 2-input AND, and constant 0(1). The edges with INVERTER attribute denote the Boolean complementation. It is easy to transform a Boolean network (function) into AIGs by simple gate transformation rules. Moreover, it is very fast to perform simulation on AIGs with respect to (w.r.t.) a large set of input vectors at one time. However, it is unlike to the BDDs which is a canonical form of Boolean functions w.r.t. a given input variable ordering. It may also have many functionally equivalent nodes in the graph. In the paper [10], SAT sweeping and structural hashing techniques were applied to reduce the graph size. More recently, Mishchenko et al. exploited SAT-based equivalent checking techniques to remove equivalent nodes while constructing an AIG, i.e., Functionally Reduced AIGs (FRAIGs) [12]. By our experimental observation, FRAIGs can represent many large Boolean functions that can not be constructed as BDDs due to the memory explosion problem.

3. DETECTING FUNCTIONAL PROPERTY

Consider a function $f(X)$ and an input $x_i \in X$. The *cofactor* of f w.r.t. x_i is $f_{x_i} = f(x_1, \cdots, x_i = 1, \cdots, x_n)$. The cofactor of f w.r.t. $\bar{x_i}$ is $f_{\bar{x_i}} = f(x_1, \cdots, x_i = 0, \cdots, x_n)$. A function f is *positive* (*negative*) *unate* in variable x_i if $f_{\bar{x_i}} \subseteq f_{x_i}$ ($f_{x_i} \subseteq f_{\bar{x_i}}$). Otherwise, it is *binate* in that variable. Given two input variables $x_i, x_j \in X$, the definitions of *non-equivalence symmetry* (*NE*), *equivalence symmetry* (*E*), and *single variable symmetry* (*SV*) of f w.r.t. x_i and x_j are summarized in Table 1.

S&S approach is also applied to check functional symmetry and unateness of target functions in our matching algorithm. Instead of enumerating all items (possible functional properties) and checking them directly, we exploit simulation to quickly remove impossible items. For those items that can not be removed by the simulation results, SAT technique is exploited to verify them. Consider a function f and some functional property p to be checked. Our detecting procedure starts by using random simulation to remove impossible items as many as possible. If there still exist some unchecked items, it repeats taking an item and checking it with SAT technique until the taken item is a true functional property of f. Guided simulation will then be used to filter out the remaining impossible items. Rather than generating pure random vectors, guided simulation will generate simulation vectors based on counter examples by SAT solving, i.e., solutions of the SAT instance.

In order to remove impossible items, we generate many pairs of random vectors (v_1, v_2)'s with Hamming distance 1 or 2 for simulation. The vector pairs with distance 1 are used to remove functional unateness and single variable symmetries, while the pairs with distance 2 are used to remove non-equivalence and equivalence symmetries. Suppose that v_1 and v_2 are disjoint on input x_i (and x_j) if their distance is 1 (2). Without loss of generality, let $f(v_1) \neq f(v_2)$ and $f(v_1) = 1$. The conditions for removing impossible functional properties and SAT-based equivalence checking are briefly summarized in Table 1, where the notation $val(v_1, x_i)$ denotes the value of x_i in the vector v_1.

4. DEFINITIONS AND NOTATIONS

Let $P = \{X_1, X_2, \cdots, X_k\}$ be a *partition* of input set X, where $\bigcup_{i=1}^{k} X_i = X$ and $X_i \cap X_j = \emptyset$ for $i \neq j$. Each X_i is an *input group* w.r.t. P. The *partition size* of P is the number of subsets X_i's in P, denoted as $|P|$. The *group size* of X_i is the number of input variables in X_i, denoted as $|X_i|$.

Definition 4.1. *Given two input sets X and Y with the same number of input variables, let $P_X = \{X_1, X_2, \cdots, X_k\}$ and $P_Y = \{Y_1, Y_2, \cdots, Y_k\}$ be two ordered input partitions of X and Y, respectively. A* ***mapping relation*** *$R = \{G_1, G_2, \cdots, G_k\}$ is a set of mappings between the in-*

put groups of P_X and P_Y, where $G_i = {X_i}^{Y_i}$ and $|X_i| = |Y_i|$. Each element $G_i \in R$ is a ***mapping group*** *which maps X_i to Y_i.* ■

DEFINITION 4.2. *Consider a mapping relation R and a mapping group $G_i = {X_i}^{Y_i}$ in R. The* ***mapping relation size*** *is the number of mapping groups in R, denoted as $|R|$. The* ***mapping group size*** *of G_i, denoted as $|G_i|$, is the group size of X_i (or Y_i), i.e., $|G_i| = |X_i| = |Y_i|$.* ■

DEFINITION 4.3. *Consider two functions $f(X)$ and $g(Y)$. Let R be a mapping relation and $G_i = {X_i}^{Y_i}$ be a mapping group in R. G_i is* ***unique*** *if and only if $|G_i| = 1$ or $X_i(Y_i)$ is a NE-symmetric set of f (g). The mapping relation R is* ***unique*** *if and only if all the mapping groups in R are unique.* ■

DEFINITION 4.4. *Let v_i be an input vector w.r.t. the input set X. The* ***input weight*** *of v_i is the number of inputs with binary value 1. It is denoted as $\rho(v_i, X)$.* ■

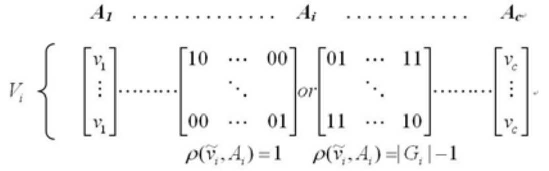
DEFINITION 4.5. *Consider a function $f(X)$ and a vector set V involving m distinct input vectors. The* ***output weight*** *of f w.r.t. V is the number of vectors v_i's in V such that $f(v_i) = 1$. It is denoted as $\sigma(f, V)$ and $0 \leq \sigma(f, V) \leq m$.* ■

5. SIMULATION APPROACH FOR DISTINGUISHING INPUTS

The idea behind our simulation approach is the same as the concept of exploiting signatures to quickly remove impossible input correspondences as many as possible. Consider two target functions $f(X)$ and $g(Y)$. Let $R = \{G_1, \cdots, G_i, \cdots, G_c\}$ be the current mapping relation. Without loss of generality, groups $G_1, \cdots, G_i$ are assumed non-unique while the remaining groups are unique. To partition a non-unique mapping group $G_i = {X_i}^{Y_i}$, for each input $x_j \in X_i$ $(y_j \in Y_i)$, we generate a vector v or a set of vectors V for simulation and use the simulation results as the signature of x_j (and y_j) w.r.t. f (and g). It can further partition G_i into two or more smaller mapping groups by means of these signatures. Suppose the mapping size of G_i is m, i.e., $|G_i| = |X_i| = |Y_i| = m$. In the following, we will propose three types of input vectors and show how to distinguish input variables of X_i (and Y_i) in terms of the simulation results.

5.1 Type-1

We first generate c subvectors $v_1, \cdots, v_i, \cdots, v_c$, where v_i is a random vector with input weight 0 or m, i.e., $\rho(v_i, X_i) = \rho(v_i, Y_i) = 0$ or m. For each input variable $x_j \in X_i$ (and $y_j \in Y_i$), the subvector $\tilde{v}_i$ with input weight 1 or $m-1$ can be obtained by complementing the value of x_j (and y_j) in v_i. The concatenated vector $v^j = v_1|\cdots|\tilde{v}_i|\cdots|v_m$ will then be used as input values for simulating on f (and g) and its corresponding output value $f(v^j)$ $(g(v^j))$ can be viewed as the signature of x_j (y_j). Fig. 1 demonstrates the vector set V_i used to partition G_i, where A_i is the set X_i or Y_i. Each vector (row) in V_i is dedicated to an input variable in X_i (Y_i). Using such a set V_i for simulation, in most cases, X_i (Y_i) can be partitioned into two subsets X_{i0} (Y_{i0}) and X_{i1} (Y_{i1}), where the signatures of input variables in these two sets are output value 0 and 1, respectively. Therefore, G_i can be divided into two mapping groups $G_{i0} = {X_{i0}}^{Y_{i0}}$ and $G_{i1} = {X_{i1}}^{Y_{i1}}$. Moreover, in our matching algorithm all non-unique mapping groups can be partitioned simultaneously.



A_1 .... A_i .... A_c
V_i
$\begin{bmatrix} v_1 \\ \vdots \\ v_1 \end{bmatrix}$ .... $\begin{bmatrix} 10 & \cdots & 00 \\ & \ddots & \\ 00 & \cdots & 01 \end{bmatrix}$ or $\begin{bmatrix} 01 & \cdots & 11 \\ & \ddots & \\ 11 & \cdots & 10 \end{bmatrix}$ .... $\begin{bmatrix} v_c \\ \vdots \\ v_c \end{bmatrix}$
$\rho(\tilde{v}_i, A_i) = 1$
$\rho(\tilde{v}_i, A_i) = |G_i| - 1$

Figure 1: Type-1 Simulation Vectors V_i of G_i.

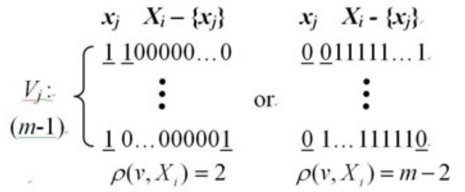


Figure 2: Type-2 Simulation Vectors V_j.

5.2 Type-2

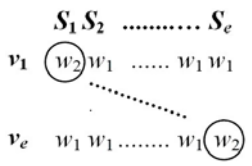
For each input $x_j \in X_i$ $(y_j \in Y_i)$, a vector set V_j involving $|G_i|-1$ vectors with weight 2 or $m-2$ will be generated. For simplicity, Fig. 2 only shows out the subvectors w.r.t. the input set X_i while the subvectors w.r.t. the remaining inputs sets can be generated like the initial subvectors v_i's of Type-1. Consider a vector in V_j. We assign 1 (or 0) to the input variable x_j and one of the remaining inputs, while the other inputs are assigned 0 (or 1). After the simulation, the output weight $\sigma(f, V_j)$ $(\sigma(g, V_j))$ will be used as the signature of x_j (y_j). The input variables with the same output weight can match to each other. Consequently, we can partition G_i into at most m groups because of $0 \leq \sigma(f, V_j) \leq m-1$. Moreover, if there exists some input $x_j \in X_i$ which can uniquely map to an input $y_j \in Y_i$, we can apply Type-1 checking to further partition the set $X_i - \{x_j\}$ $(Y_i - \{y_j\})$ using the simulation results for the vector set V_j.

5.3 Type-3

The third type of vectors can only be used for the mapping group $G_i = {X_i}^{Y_i}$, where X_i and Y_i involves several *NE-symmetric* sets. The idea is mainly based on functional symmetry that function f is invariant under the permutation of inputs in its NE-symmetric set. Suppose X_i consists of e symmetric set $S_1, S_2, \cdots, S_e$ each with k input variables. To partition the group G_i, two random vectors a_1 and a_2 with different input weight w_1 and w_2 will be generated, where $0 \leq w_1, w_2 \leq k$. For each symmetric set S_i, we then generate a vector v_i by assigning a_2 and a_1's to S_i and the remaining sets, respectively. Fig. 3 shows only the weight distribution of vectors v_i's, where v_i is dedicated to S_i for simulation. As to the other mapping groups, the weights of their subvectors must be 0 or $|G_i|$ like we used in Type-1. Using such a vector set for simulation, we can partition $S_1, S_2, \cdots, S_e$ into two groups of symmetric sets which have output simulation value 0 and 1, respectively. It is easy to show that at most $k \times (k+1)$ combinations of w_1 and w_2 are required for this type of checking.

Our matching algorithm checks that $f(X)$ and $g(Y)$ can not match to each other using the following observation.

OBSERVATION 5.1. *Consider a non-unique mapping group $G_i = {X_i}^{Y_i}$. Through the simulation and checking steps as we described above, let P_{X_i} and P_{Y_i} be the resultant partitions w.r.t. to X_j and Y_j, respectively. For each group $A \in P_{X_i}$, there is a corresponding group $B \in P_{Y_i}$. The following shows two situations that $f(X)$ and $g(Y)$ can not match to each other:*



$S_1\, S_2$ .... S_e
v_1
w_2 w_1 ...... w_1 w_1
v_e
w_1 w_1 .... w_1 w_2

Figure 3: Type-3 Simulation Vectors of G_i.

- $|P_{X_j}| \neq |P_{Y_j}|$, *i.e., their partition sizes are different.*
- $|A| \neq |B|$, *i.e., their set sizes are different.* ■

6. S&S-BASED MATCHING ALGORITHM

6.1 Our Matching Algorithm

Our S&S-based Boolean matching algorithm shown in Fig. 4 can match two target functions $f(X)$ and $g(Y)$ under a *threshold* bound, i.e., the maximum number of simulation rounds. It is mainly divided into three phases: *Initialization*, *Simulation*, and *Recursion*. In the *Initialization* phase, it exploits functional symmetries and unateness to initialize the mapping relation R followed by computing the maximum number of mapping groups in R, denoted as $MaxSize$. Since the input variables in an NE-symmetric set can be permuted without affecting the functionality, $MaxSize$ is equal to the number of NE-symmetric sets adding the number of non-symmetric inputs. It is the upper bound used to check the first terminating condition of the second phase. In the *Simulation* phase, it improves the mapping relation R by the *Simulate-and-Update* procedure implementing the simulation approach described in Section 5. This step is repeated until it can find a unique mapping relation R with $|R| = MaxSize$ or no improvement can be made to R under the *threshold* bound. If this phase is ended by the second terminating condition, it calls the *Recursive-Matching* procedure and enters into the *Recursion* phase to search the final mapping relation. Otherwise, the *SAT-Verify* procedure exploiting SAT technique is called to verify if two target functions are matched under the unique mapping relation R searched by the *Simulation* phase.

6.2 Recursive Matching Algorithm

Fig. 5 shows our recursive matching algorithm. Consider target functions f, g, a mapping relation R, and the size bound $MazSize$ of R. It starts by checking if R is unique and so to verify R using SAT technique. For the case that R is not unique, the smallest non-unique mapping group $G_i = {X_i}^{Y_i}$ in R will be selected to be partitioned. Consider the mapping of an input $x_j \in X_i$ to an input $y_k \in Y_i$. It will partition G_i into two mapping groups $A = \{x_j\}^{\{y_k\}}$ and $B = (X_i - \{x_j\})^{Y_i - \{y_k\}}$. With such a partitioning, we can derive a new mapping relation $tmpR = R \cup T - \{G_i\}$ with mapping relation size $|R| + 1$, where $T = \{A, B\}$ is derived from G_i. *Simulate-and-Update* procedure is then called to further improve $tmpR$ and return a new mapping relation $NewR$. If $NewR$ is not empty, it will call itself again; otherwise, it indicates a wrong selection of input mapping. In addition, this algorithm can be easily modified to find all feasible mapping relations as shown in Fig. 5.

6.3 Implementation Issues

6.3.1 Control of Random Vector Generation

By our experimental results, most of the runtime was consumed by the simulation phase for some benchmarking circuits. The reason is that too many random vectors gener-

```
Algorithm S&S-Boolean-Matching($f(X), g(Y), threshold$)
Input: $f$ and $g$ are target functions;
        $threshold$: the maximum # of simulation rounds;
Output: $\emptyset$ or $\psi$, i.e., the feasible mapping relation of $f$ and $g$;
Begin
    $\psi = \emptyset$;   $cnt = 0$;
    $R$ = Initial-Mapping($f, g$);   // Phase 1
    $MaxSize$ = #NE_Classes + #Non_Symm_Inputs;
    while ($|R| < MaxSize$ and $cnt < threshold$) do
        $NewR$ = Simulate-And-Update($f, g, R$);   // Phase 2
        if ($NewR = \emptyset$) return $\emptyset$;
        if ($|R| == |NewR|$) then
            $cnt = cnt + 1$;   // no improvement made on $R$
        else
            $R = NewR$;   $cnt = 0$;
        endif
    endwhile
    if ($|R| < MaxSize$) then
        $\psi$ = Recursive-Matching($f, g, R, MaxSize$);   // Phase 3
    else
        if (SAT-Verify($f, g, R$) is TRUE) $\psi = R$;
    endif
    return $\psi$;
End
```

Figure 4: Our Boolean Matching Algorithm.

ated for simulation are useless to improve current mapping relation. Thus it will incur a large amount of iterations on the simulating and updating steps in this phase. Instead of generating random vectors without using any criterion, we propose a simple heuristic to control the generation of two adjacent random vectors. Let v_1 be the first random vector. The second vector v_2 can then be generated by randomly complementing $n/2$ inputs in v_1, where n is the number of input variables of target functions. We expect it can evenly distribute the random vectors in the Boolean space and so that it can quickly converge to find feasible mapping relation. Our experimental result shows the runtime can be greatly reduced for some circuits.

6.3.2 Reduction of Simulation Time

Our matching algorithm can be easily extended to deal with Boolean functions with multiple outputs. While matching two target functions with multiple outputs, we can reduce the simulation time by utilizing the mapping relation found so far. Clearly, the more the unique mapping groups we find, the less the number of outputs with indistinguishable inputs is. So, rather than simulating the whole Boolean network, simulating the subnetwork involving these outputs and their transitive fanin nodes is enough. Our experimental result reveals that our matching algorithm can reduce the runtime significantly as it approaches the end of matching process.

6.3.3 Analysis of Space Complexity

During the simulation process, we need to store the simulation vectors for all nodes in the Boolean network. Let the number of inputs and number of nodes in the Boolean network be I and N, respectively. The memory space used by our matching algorithm is $M \times I \times N$ words (4 bytes), where M is an adjustable parameter, i.e., the number of sets used in each simulation round. The smaller M means that it can reduce the storage space and simulation time in a simulation round. On the contrary, it may need more simulation rounds to improve the mapping relation. Besides,

```
**Algorithm** $Recursive\text{-}Matching(f(X), g(Y), R, MaxSize)$
**Input:** $f$ and $g$ are target functions;
        $R$: the current mapping relation;
        $MaxSize$: the maximum size of mapping relation $R$;
**Output:** $\emptyset$ or $\psi$, i.e., the feasible mapping relation of $f$ and $g$;
**Begin**
    **if** $(|R| == MaxSize)$ **then**     // the terminating condition
        **if** $(SAT\text{-}Verify(f, g, R)$ is **TRUE**) **return** $R$;
        **return** $\emptyset$;
    **endif**
    $G_i$ = the smallest *non-unique mapping group* in $R$;
    $\psi = \emptyset$; Choose an input $x_j \in X_i$;
    **for** each possible mapping relation $T$ w.r.t. $G_i$ **do**
        $tmpR = R \cup T - \{G_i\}$;
        $NewR = Simulate\text{-}And\text{-}Update(f, g, tmpR)$;
        **if** $(NewR \neq \emptyset)$
            $\psi = Recursive\text{-}Matching(f, g, NewR, MaxSize)$;
        // comment the next line if to find all solutions
        **if** $(\psi \neq \emptyset)$ **return** $\psi$;
    **endfor**
    **return** $\psi$;
**End**
```

Figure 5: Recursive Matching Algorithm.

Table 2: Matching Results for Threshold 1000

#Input	#Circuit	#Solved	CPU Time (sec.)		
			Min	Avg	Max
4∼10	31	31	0.00	0.04	0.30
11∼20	21	21	0.01	0.55	8.04
21∼30	14	14	0.03	0.21	1.29
31∼40	10	9	0.07	1.16	4.79
41∼50	8	8	0.22	2.94	5.83
51∼257	28	26	0.14	2.57	17.56

it may stop the simulation phase early, and thus enter into the recursive matching phase. If there exist many large non-unique mapping groups, then the runtime will increase significantly because it may incur a large amount of simulation and SAT verification on infeasible mapping relations.

7. EXPERIMENTAL RESULTS

The proposed S&S-based Boolean matching algorithm had been implemented into Berkeley's ABC system on the Linux platform with dual Intel Xeon 3.0 GHz CPU's. To demonstrate the efficiency of our algorithm, MCNC and LGSyn benchmarking sets were tested in our experiments. For each tested circuit, we randomly permuted its input variables to generate a new circuit for being matched. In addition, to make our experimental results more convincible, we restructured this new circuit by executing a simple script file including some synthesis commands offered by ABC. Two sets of experiments were conducted to test our matching algorithm.

The first experiment was conducted to search all feasible mapping relations on 112 circuits with input number ranging from 4 to 257. The experimental results showed that three circuits C6288, i3, and o64 can not be solved within 5000 seconds. To dissect these circuits, we found that one of the two mapping relations of C6288 can not be verified by SAT technique while the other two circuits have a large amount of feasible mapping relations because they owns a great many $\mathcal{G}$-*symmetries* [11]. If only to search one feasible mapping, the execution times for i3 and o64 are 0.49 sec. and 8.67 sec., respectively. The experimental results were summarized in Table 2 w.r.t. the circuit input size. In this table, the first two columns show the input ranges of benchmarking circuits and number of circuits in different input ranges, respectively.

Table 3: Comparison on the Effects of Three Phases

	Functional Property(1)			+Sim.(2)	+Rec.(3)
	+Unate	+Symm	+SVS		
#Circuit	19	49	71	94	109
#Inc	19	30	22	23	15
ratio (%)	17.4	44.9	65.1	86.2	100

Table 4: The Results of Circuits with Inputs > 50

Circuit	#I	#O	#Sol		CPU Time (sec.)		
			O	S	Orig	Unate	+Symm
apex3	54	50	1	1	0.10	0.10	0.38
apex5	117	88	144	1	7.11	2.96	0.68
apex6	135	99	2	1	1.86	0.42	0.33
C2670	233	140	-	2	*	*	7.96
C5315	178	123	4	1	6.31	2.86	3.29
C7552⋆	207	108	-	1	*	*	14.56
C880	60	26	8	1	0.28	0.20	0.25
dalu	75	16	2	1	1.20	3.36	5.47
des	256	245	1	1	10.21	0.25	2.33
e64	65	65	1	1	0.01	0.79	0.32
ex4p	128	28	-	4096	*	*	6.08
example2	85	66	1	1	0.05	0.02	0.23
frg2	143	139	1	1	0.45	0.10	0.72
i10	257	224	48	2	25.63	15.16	17.56
i2	201	1	-	1	*	*	1.02
i4	192	6	-	1	*	*	0.22
i5	133	66	1	1	0.18	0.03	0.35
i6	138	67	1	1	0.50	0.02	0.14
i7	199	67	1	1	0.82	0.04	0.19
i8	133	81	1	1	0.57	0.06	0.40
i9	88	63	1	1	0.18	0.03	0.16
pair	173	137	1	1	0.84	0.64	2.44
rot	135	107	72	1	3.79	1.69	1.25
x1	51	35	2	1	0.17	0.13	0.14
x3	135	99	2	1	2.05	0.28	0.32
x4	94	71	2	1	0.62	0.37	0.15
Total					> 25063	> 25029	66.94
Avg					> 964	> 963	2.57

-: unknown   *: CPU time > 5000 sec.   ⋆: memory explosion

The third column labeled **#Solved** shows the number of circuits solved by our matching algorithm. The next three columns **Min**, **Avg**, and **Max** show the minimum, average and maximum runtime for the solved circuits, respectively. It shows that our algorithm is very efficient for the circuits with moderate to large input sets.

For those solved circuits, we also compared the effectiveness of three phases and the comparison results were shown in Table 3. The rows labeled **#Circuit** and **#Inc** show the number of circuits that have been matched successfully and the number of increased matched circuits in each individual step, respectively. In the first phase, we compared the effect of incrementally applying different functional properties. The columns named as **+Unate**, **+Symm**, and **+SVS** show the results of only using unateness, adding E&NE symmetry, and adding SV symmetry, respectively. It is clear that the more functional properties are used, the more circuits can be solved in the first phase. It shows that 71 and 94 circuits can be solved after the first phase and the second (simulation) phase, respectively. All the remaining circuits can be solved by the third (recursion) phase.

Table 4 shows the experimental results for those circuits with input size greater than 50. The first three columns labeled **Circuit**, **#I**, and **#O** show the circuit name, number of input variables, and number of outputs in this benchmarking circuit, respectively. The next two columns **O** and **S** are the numbers of mapping relations found by our matching algorithm without using and using functional properties. It should be noted that the solutions induced by NE-symmetry is not taken into account on the numbers shown in the **S** column. Moreover, the numbers greater than one indicate that these benchmarking circuits own $\mathcal{G}$-*symmetry.* The last three columns named as **Orig**, **Unate**, and **+Symm** compare the execution times of without using functional property, using only functional unateness, and adding functional symmetries, respectively. The result shows that there are

Table 5: Benchmarking Results for s-Series Circuits

Circuit	#I	#O	#N	#BDD	Symmetry	#Sol	CPU Time (sec.)						
							Orig		Unate		+Symm		
							First	All	First	All	First	All	SAT
s4863 †	153	120	3324	56691	1(8),1(9)	$4 \cdot 8! \cdot 9!$	*	*	2.56	*	1.87	1.88	0.01
s3384	264	209	2720	882	22(2)	2^{22}	4.79	*	2.14	*	4.02	4.02	0.00
s5378	199	213	2850	⋆	4(2),1(5),1(7)	$(2!)^4 \cdot 5! \cdot 7!$	1.31	*	3.38	*	2.42	2.42	0.00
s6669 †	322	294	4978	22957	32(2), 1(17)	$4 \cdot 2^{32} \cdot 17!$	6.30	*	2.83	*	4.08	50.54	46.50
s9234.1	247	250	4023	4545	-	1	3.41	3.41	5.84	5.84	7.82	7.82	0.00
s38584.1	1464	1730	26702	22232	1(3),1(9)	$3! \cdot 9!$	76.31	*	210.13	*	457.82	457.82	0.03
s38417	1664	1742	23308	55832099	2(2),1(3)	$(2!)^2 \cdot 3!$	91.81	*	324.57	*	998.53	998.55	0.13
Total							183.93		551.45		1476.56	1523.05	46.67
Avg							30.66		78.78		210.94	217.58	6.67
Ratio							0.15		0.37		1.00	1.03	0.03

†: circuits own $\mathcal{G}$-*symmetry* -: no symmetry $m(n)$: m NE-symmetric sets with n inputs *: CPU time > 5000 sec. ⋆: memory explosion

5 out of 26 circuits can not be solved by our matching algorithm without using full functional property within 5000 seconds. The reason why this situation occurs is that these circuits have a great many NE-symmetries. However, it can resolve all cases if functional symmetries are utilized to reduce the searching space. The average runtime of using full functional property is 2.57 second. It clearly reveals that our S&S-based matching algorithm is indeed effective and efficient for solving the Boolean matching problem. In this experiment, the BDD's of these circuits were also built for comparison with AIGs. It shows the circuit C7752 had the memory explosion problem while constructing BDD without using dynamic ordering.

In order to test our matching algorithm on very large Boolean networks, the second experiment was conducted to test the circuits in ISCAS89 benchmarking set. Since these circuits are sequential, we executed the *comb* command in ABC to transform them into combinational circuits. Table 5 shows the partial experimental results. For each circuit, the columns **#N**, **#BDD**, **Symmetry**, and **#Sol** show the number of nodes in the Boolean network (AIG), number of nodes in the constructed BDD, NE-symmetry, and number of feasible mappings. The CPU times of finding the first feasible mapping, finding all feasible mapping relations, and performing SAT verification of two target circuits are shown in the **First**, **All**, and **SAT** columns, respectively. The experimental results show that our algorithm can not find all feasible mappings for those circuits with a great many NE-symmetries unless we detect them in advance. It also reveals that only searching the first feasible mapping relation without using symmetry is faster than the one using symmetry in some cases. The reason why this situation occurs is that it took too much time on detecting symmetries for the circuits. The result also shows only a very small amount of runtime was consumed by SAT verification for all circuits except the s6669 circuit. Besides, the AIG size was far less than the BDD size in many tested circuits and the s5378 circuit had the memory explosion problem. In summary, our S&S-based Boolean matching algorithm can be easily adjusted to fulfill different requirements for large Boolean networks.

8. CONCLUSIONS

We had proposed a S&S-based Boolean matching algorithm in this paper. Three types of input vectors were generated for simulation and their simulation results were used to distinguish the inputs of two target functions. Our matching algorithm had been tested on a set of large benchmarking circuits. The experimental results reveal that our algorithm is indeed effective and efficient for solving the Boolean matching problem on very large functions. The future work will be dynamically adjusting threshold value in the simulation process and extending our matching algorithm to deal with input/output phase assignment.

9. REFERENCES

[1] A. Mishchenko, S. Chatterjee, R. Brayton, and N. En, "Improvements to Combinational Equivalence Checking," in *Proc. of Internatioanl Conference on Computer-Aided Design*, pp. 836-843, Nov. 2006.

[2] S. Plaza, K. Chang, I. Markov, and V. Bertacco, "Node Mergers in the Presence of Don't Cares," in *Proc. of Asia and South Pacific Design Automation Conference*, pp. 414-419, Jan. 2007.

[3] Qi Zhu, N. Kitchen, A. Kuehlmann, A. Sangiovanni-Vincentelli, "SAT Sweeping with Local Observability Don't-Cares," in *Proc. of Design Automation Conference*, pp. 229-234, July 2006.

[4] A. Mishchenko, et. al, "Using Simulation and Satisfiability to Compute Flexibilities in Boolean Networks," *IEEE Transaction on Computer-Aided-Design of Integrated Circuits and Systems*, Vol. 25, No. 5, pp. 743-755, May 2006.

[5] L. Benini and G. De Micheli, "A Survey of Boolean Matching Techniques for Library Binding," *ACM Trans. on Design Automation of Electronic Systems*, Vol. 2, No. 3, pp. 193-226, July 1997.

[6] Afshin Abdollahi, "Signature Based Boolean Matching in the Presence of Don't Cares," in *Proc. of Design Automation Conference*, pp. 642-647, June 2008.

[7] G. Agosta, et. al, "A Unified Approach to Canonical Form-based Boolean Matching," in *Proc. of Design Automation Conference*, pp. 841-846, June 2007.

[8] A. Abdollahi and M. Pedram, "A New Canonical Form for Fast Boolean Matching in Logic Synthesis and Verification," in *Proc. of Design Automation Conference*, pp. 379-384, June 2005.

[9] K.H. Wang and Chung-Ming Chan, "Incremental Learning Approach and SAT Model for Boolean Matching with Don't Cares," in *Proc. of International Conference on Computer-Aided Design*, pp. 234-239, November 2007.

[10] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ranai, "Robust Boolean Reasoning for Equivalence Checking and Functioanl Property Verification," *IEEE Transaction on Computer-Aided-Design of Integrated Circuits and Systems*, Vol. 21, No. 12., pp. 1377-1394, Dec. 2002.

[11] J. Mohnke, P. Molitor, and S. Malik, "Limits of Using Signatures for Permutation Independent Boolean Comparison," in *Proc. of ASP Deisgn Automaiton Conference*, pp. 459-464, 1995.

[12] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGS: A Unifying Representation for Logic Synthesis and Verification," *ERL Technical Report*, EECS Dept., UC Berkeley, March 2005.

[13] J. Marques-Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Transactions on Computer-Aided-Design of Integrated Circuits and Systems*, Vol. 48, No. 5, pp. 506-521, May 1999.

[14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. of Design Automation Conference*, pp. 530-535, June 2001.

[15] J. P. Marques-Silva and K. A. Sakallah, "Boolean Satisfiability in Electronic Design Automation," in *Proc. of Design Automation Conference*, pp. 675-680, June 2000.