Large-scale Boolean Matching

Hadi Katebi and Igor L. Markov
University of Michigan, 2260 Hayward St., Ann Arbor, MI 48109
{hadik, imarkov}@eecs.umich.edu

1 Chapter Overview

In this chapter, we propose a methodology for Boolean matching under permutations of inputs and outputs (PP-equivalence checking problem) — a key step in incremental logic design that identifies large sections of a netlist that are not affected by a change in specifications. When a design undergoes incremental changes, large parts of the circuit may remain unmodified. In these cases, the original and the slightly modified circuits share a number of functionally equivalent sub-circuits. Finding and reutilizing the equivalent subcircuits reduces the amount of work in each design iteration and accelerates design closure. In this chapter, we present a combination of fast and effective techniques that can efficiently solve the PP-equivalence checking problem in practice. Our approach integrates graph-based, simulation-driven and SAT-based techniques to make Boolean matching feasible for large circuits. We verify the validity of our approach on ITC'99 benchmarks. The experimental results confirm scalability of our techniques to circuits with hundreds and even thousands of inputs and outputs.

The remainder of this chapter is organized as follows. Section 2 describes the motivation of this work. Section 3 provides relevant background and discusses previous work on Boolean matching. Section 4 gives an overview of proposed signature-based techniques. Section 5 describes our SAT-based matching approach. Section 6 validates our method in experiments on available benchmarks, and Section 7 summarizes our work.

2 Introduction

Boolean matching is the problem of determining whether two Boolean functions are functionally equivalent under the permutation and negation of inputs and outputs.

This formulation is usually referred to as the *generalized* Boolean matching problem or PNPN-equivalence checking (PNPN stands for Permutation and Negation of outputs and Permutation and Negation of inputs); however, different variants of the problem have been introduced for different synthesis and verification applications. The matching problem that we discuss in this chapter is PP-equivalence checking: two Boolean functions are called PP-equivalent if they are equivalent under permutation of inputs and permutation of outputs. The simplest method to determine whether two n-input m-output Boolean functions are PP-equivalent is to explicitly enumerate all the $m!n!$ possible matches and perform tautology checking on each. However, this exhaustive search is computationally intractable.

PP-equivalence checking finds numerous applications in verification and logic synthesis. In many cases, an existing design is modified incrementally leaving a great portion of the design untouched. In these cases, large isomorphic sub-circuits exist in original and slightly modified circuits [15]. Identifying such sub-circuits and reutilizing them whenever possible saves designers a great amount of money and time. Due to the fact that modifications to the original circuit are introduced by changing certain specifications and the fact that even a slight change in specifications can lead to large changes in implementation [9], PP-equivalence checking helps designers identify isomorphic and other equivalent sub-circuits.

Specifically, PP-equivalence checking can be used to find the minimal set of changes in logic, known as *logic difference*, between the original design and the design with modified specification. DeltaSyn [10] is a tool developed at IBM Research that identifies and reports this logic difference. The current version of DeltaSyn uses a relatively inefficient and unscalable Boolean matcher that only exploits the symmetry of inputs to prune the search space.

Incremental Sequential Equivalence Checking (SEC) is another application of PP-equivalence checking where isomorphic sub-circuits can be used to create a number of highly-likely candidate equivalent nodes [15]. The current implementation of incremental SEC tires to find isomorphic subgraphs by performing extended simulation and finding structural similarities. Although the Boolean approach presented in our paper does not fully exploit the structural similarities between two circuits, we believe that our techniques combined with structural verification techniques create a much more powerful tool for detecting isomorphic subgraphs.

Motivated by the practical importance of PP-equivalence checking in many EDA applications, we develop fast and scalable Boolean matching algorithms and implement them in the ABC package — an established system for synthesis and verification of combinational and sequential logic circuits [12]. The collection of all these techniques creates a powerful Boolean matching module that can be integrated into Combinational Equivalence Checking (CEC) to enhance its functionality. To this end, CEC requires two designs whose primary I/Os match by name. Our work allows one to relax this requirement with the help of a Boolean matcher. We call the new command *Enhanced CEC* (*ECEC*). Figure 1 shows how our Boolean matcher is integrated with CEC.

In general, algorithms for Boolean matching fall into two major categories: signature-based and canonical form based. A signature is a property of an input

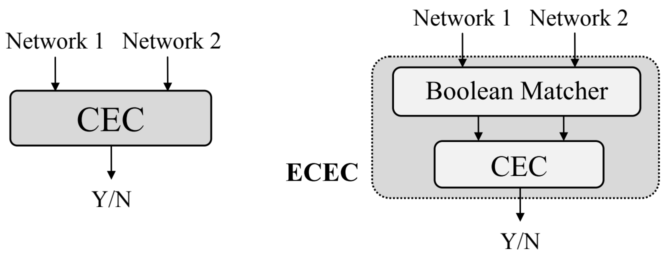


Fig. 1 The CEC (pre-existing) and ECEC (our) flows.

or an output that is invariant under permutations and negation of inputs. The goal of signature-based matching is to prune the Boolean matching space by filtering out impossible I/O correspondences [4, 1]. On the other hand, in matching based on canonical forms, first canonical representations of two Boolean functions are computed and then compared against each other to find valid I/O matches [3, 2]. Here, our PP-equivalence checking method first prunes the search space using graph algorithms and simulation signatures, then it invokes SAT-solving until exact I/O matches are found (Figure 2).

Main contributions of our work include:

1. **Analyzing functional dependency**. In a Boolean network with multiple outputs, some inputs affect only a fraction of the outputs, and different outputs are affected in different ways. Hence, by analyzing the functional dependency of outputs on inputs, we can distinguish the I/Os.
2. **Exploiting input observability and output controllability**. We use the observability of inputs and the controllability of outputs as (1) effective matching signatures, and (2) ordering heuristics for our SAT-based matching.
3. **Building a SAT-tree**. When information about controllability, observability, and all simulation-based information are exhausted, we resort to SAT-solving and optimize the efficiency of SAT calls. This is accomplished through the concept of a SAT-tree, which is pruned in several ways.
4. **Pruning SAT-tree using SAT counterexamples**. In our SAT-based matching, the SAT-solver returns a counterexample whenever it finds an invalid match. The information in these counterexamples is then used to prune the SAT-tree.

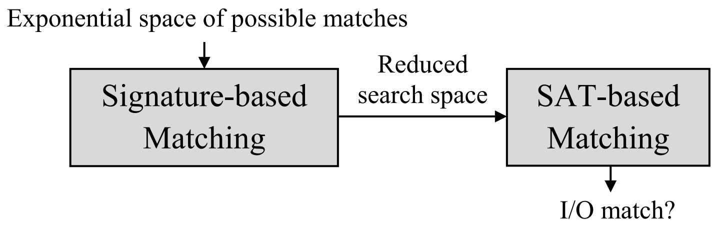


Fig. 2 Overview of our proposed Boolean matching approach.

3 Background and Previous Work

In this section, we first review some common definitions and notation. Then, we explain the *And-Inverter* representation of Boolean networks and we compare its usage to that of conventional *Binary Decision Diagrams*. Next, we discuss *Boolean satisfiabilty* and explore its role in combinational equivalence checking. Relevant work in Boolean matching is reviewed at the end of this section.

3.1 Definitions and Notation

In the following definitions, an *input set* of a Boolean network N refers to the set of all the inputs of N. Similarly, an *output set* of N refers to the set of all the outputs of N. An I/O set is either an input set or an output set.

Definition 1. A *partition* $P_x = \{X_1,...,X_k\}$ of an I/O set $X = \{x_1,...,x_n\}$ is a collection of subsets $X_1,...,X_k$ of X such that $\cup_{i=1}^{k} X_i = X$ and $X_i \cap X_j = \emptyset$ for all $i \neq j$. *Partition size* of P_x is the number of subsets in P_x and is denoted by $|P_x|$. Each X_i in P_x is called an I/O *cluster* of P_x. The *cardinality* of X_i, denoted by $|X_i|$, is the number of I/Os in X_i.

Definition 2. A partition $P_x = \{X_1,...,X_k\}$ of set X is an *ordered partition* if the subsets $X_1,...,X_k$ are *totally ordered*, i.e., for any two subsets X_i and X_j, it is known whether $X_i < X_j$ or $X_j < X_i$.

Definition 3. Two ordered partitions $P_x = \{X_1,...,X_k\}$ of set X and $P_y = \{Y_1,...,Y_k\}$ of set Y are *isomorphic* if and only if $|P_x| = |P_y| = k$ and $|X_i| = |Y_i|$ for all i, and *non-isomorphic* otherwise. Two isomorphic partitions are called *complete* if and only if $|X_i| = |Y_i| = 1$ for all i.

Definition 4. The *positive cofactor* of function $f(x_1,..,x_n)$ with respect to variable x_i, denoted by f_{x_i}, is $f(x_1,..,x_i = 1,...,x_n)$. Similarly, the *negative cofactor* of $f(x_1,..,x_n)$ with respect to variable x_i, denoted by $f_{x_i'}$, is $f(x_1,..,x_i = 0,...,x_n)$.

Definition 5. A function $f(x_1,...,x_n)$ is *positive unate* in variable x_i if and only if the negative cofactor of f with respect to x_i is covered by the positive cofactor of f with respect to x_i, i.e., $f_{x_i'} \subseteq f_{x_i}$. Likewise, f is *negative unate* in variable x_i if and only if $f_{x_i} \subseteq f_{x_i'}$. f is called *binate* in x_i if it is not unate in it.

3.2 And-Inverter Graphs (AIGs)

Recent tools for scalable logic synthesis, e.g., ABC, represent Boolean functions using the *And-Inverter Graph* (*AIG*) data structure. An AIG is a Boolean network composed of two-input AND gates and inverters. *Structural hashing* of an AIG is a transformation that reduces the AIG size by partially canonicalizing the AIG structure [13]. Representing a Boolean function in its AIG form is preferable to its *Binary Decision Diagram* (*BDD*) form mainly because AIGs result in smaller space complexity. Also, functional simulation can be performed much faster on AIGs, but AIGs are only locally canonical.

3.3 Boolean Satisfiability and Equivalence-checking

Boolean Satisfiability (*SAT*) is the problem of determining whether there exists a variable assignment to a Boolean formula that forces the entire formula evaluate to true; if such an assignment exists, the formula is said to be *satisfiable* and otherwise *unsatisfiable*. Pioneering techniques developed to solve the SAT problem were introduced by Davis, Putnam, Logemann and Loveland in early 1960s. They are now referred to as DPLL algorithm [7, 6]. Modern SAT solvers, such as MiniSAT [8], have augmented DPLL search by adding efficient *conflict analysis*, *clause learning*, *back-jumping* and *watched literals* to the basic concepts of DPLL.

SAT is studied in a variety of theoretical and practical contexts, including those arising in EDA. CEC is one of the main applications of SAT in EDA. If two single-output Boolean functions f and g are equivalent, then $f \bigoplus g$ must always evaluate to 0, and vice versa. Now, instead of simulating all input combinations, we take advantage of SAT solvers: if $f \bigoplus g$ is *unsatisfiable*, then $f \bigoplus g$ is zero for all input combinations and hence f and g are equivalent; and if $f \bigoplus g$ is *satisfiable*, then f and g are not equivalent and the satisfying assignment found by the SAT-solver is returned as a counterexample. $f \bigoplus g$ is called the *miter* of f and g [14]. If f and g have more than one output, say m outputs $f_1, \ldots, f_m$ and $g_1, \ldots, g_m$, $M_i = f_i \bigoplus g_i$ is first computed for all i and then $M_1 + \ldots + M_m$ is constructed as the miter of f and g. In our approach, instead of building one miter for the entire circuit and handing it off to the SAT solver, we try to find equivalent intermediate signals by simulation, and use SAT to prove their equivalence. Counterexamples from SAT are used to refine simulation.

3.4 Previous Work

Research in Boolean matching started in the early 1980s with main focus on technology mapping (cell binding). A survey of Boolean matching techniques for library binding is given in [4]. Until recently, Boolean matching techniques scaled only to 10-20 inputs and one output [5, 2], which is sufficient for technology mapping, but not for applications considered in our work. In 2008, Abdollahi and Pedram presented algorithms based on canonical forms that can handle libraries with numerous cells limited to approximately 20 inputs [2]. Their approach uses generalized signatures (signatures of one or more variables) to find a canonicity-producing (CP) phase assignment and ordering for variables.

A DAC 2009 paper by Wang, Chan and Liu [17] offers simulation-driven and SAT-based algorithms for checking P-equivalence that scale beyond the needs of technology mapping. Since our proposed techniques also use simulation and SAT to solve the PP-equivalence checking problem, we should articulate the similarities and the differences. Firstly, we consider the more general problem of PP-equivalence checking where permutation of outputs (beside permutation of inputs) is allowed. In PP-equivalence, the construction of miters must be postponed until the outputs are matched, which seems difficult without matching the inputs. To address this challenge, we develop the concept of SAT-tree which is pruned to moderate the overall runtime of PP-matching. In addition to our SAT-based approach, we also use graph-based techniques in two different ways: to initially eliminate impossible I/O correspondences and to prune our SAT-tree. Furthermore, we have implemented three simulation types; two as signatures for outputs (type 1 and type 3) and one as a signature for inputs (type 2). While our type-2 simulation is loosely related to one of the techniques described in [17], the other two simulations are new. We additionally introduce effective heuristics that accelerate SAT-based matching.

4 Signature-based Matching Techniques

We now formalize the PP-equivalence checking problem and outline our Boolean matching approach for two *n*-input *m*-output Boolean networks.

Definition 6. Consider two I/O sets X and Y of two Boolean networks N_1 and N_2 with their two isomorphic ordered partitions $P_x = \{X_1, \ldots, X_k\}$ and $P_y = \{Y_1, \ldots, Y_k\}$. A *cluster mapping* of X_i to Y_i, denoted by $X_i \mapsto Y_i$, is defined as the mapping of I/Os in X_i to all possible permutations of I/Os in Y_i. A *mapping* of X to Y with respect to P_x and P_y, denoted by $X \mapsto Y$, is defined as mapping of all same-index clusters of X and Y, i.e., $X_i \mapsto Y_i$ for all i. $X \mapsto Y$ is called a complete mapping if P_x and P_y are complete.

Given two input sets X and Y and two outputs sets Z and W of two Boolean networks N_1 and N_2, the goal of PP-equivalence checking is to find two complete mappings $X \mapsto Y$ and $Z \mapsto W$ such that those mappings make N_1 and N_2 behave functionally the same. In order to accomplish this, we first partition or *refine* these I/O sets based on some total ordering criteria. This so-called signature-based matching allows us to identify and eliminate impossible I/O matches. After this phase, we rely on SAT-solving to find the two complete mappings. Furthermore, Definition 6 implies the following lemma.

Lemma 1. *If at any point in the refinement process of two I/O sets X and Y, P_x and P_y become non-isomorphic, we conclude that N_1 and N_2 behave differently and we stop the Boolean matching process.*

As mentioned earlier, refinement at each step requires a total ordering criterion, tailored to the specific refinement technique used. Therefore, whenever we introduce a new matching technique, we also explain its ordering criterion. Furthermore, the following techniques are applied to the two input circuits one after another.

4.1 Computing I/O Support Variables

Definition 7. Input x is a *support variable* of output z and output z is a *support variable* of input x, if there exists an input vector V such that flipping the value of x in V flips the value of z.

Definition 8. The *support* of input (or output) x, denoted by $Supp(x)$, is the set of all the support variables of x. The *cardinality* of the support of x, denoted by $|Supp(x)|$, is the number of I/Os in $Supp(x)$. The *degree* of x, denoted by $D(x)$, is defined as the cardinality of its support.

The goal here is to find outputs that might be functionally affected by a particular input and inputs that might functionally affect a particular output. Here, we contrast *functionally* matching with *structurally* matching in the sense that two structurally different circuits with the same functionality should have the same I/O support. In general, the lack of structural dependency between an output and an input precludes a functional dependency, and the presence of a structural dependency most often indicates a functional dependency — this can usually be confirmed by random simulation, and in rare cases requires calling a SAT-solver [11].

Example 1. Consider a 4-bit adder with input set $X = \{C_{in}, A_0, ..., A_3, B_0, ..., B_3\}$ and output set $Z = \{S_0, ..., S_4\}$. The ripple-carry realization of this adder is shown in Figure 3.

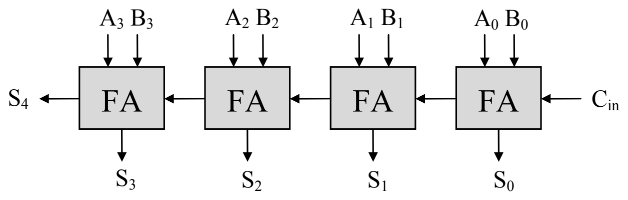


Fig. 3 4-bit ripple-carry adder.

It is evident from the above circuit that A_0 can affect the values of $S_0, ..., S_4$ and A_1 can affect the value of $S_1, ..., S_4$. Hence, $Supp(A_0) = \{S_0, ..., S_4\}$ and $Supp(A_1) = \{S_1, ..., S_4\}$. Similarly, the value of S_0 is only affected by the value of A_0, B_0 and C_{in}. Hence, $Supp(S_0) = \{A_0, B_0, C_{in}\}$.

4.2 Initial refinement of I/O clusters

Lemma 2. *Two inputs (outputs) can match only if they have the same degree.*

Taking advantage of Lemma 2, we can initially refine the I/O sets by gathering all I/Os of the same degree in one subcluster and then sort the subclusters based on the following ordering criterion:

Ordering criterion 1. Let i and j be two inputs (outputs) with different degrees and assume that $D(i) < D(j)$. Then, the subcluster containing i precedes the subcluster containing j.

Example 2. Consider the 4-bit adder of Example 1. The degree of each input and output is given below:
$D(A_0) = D(B_0) = D(C_{in}) = 5$
$D(A_1) = D(B_1) = 4$
$D(A_2) = D(B_2) = 3$
$D(A_3) = D(B_3) = 2$
$D(S_0) = 3$
$D(S_1) = 5$
$D(S_2) = 7$
$D(S_3) = D(S_4) = 9$
The ordered partitions of the I/O sets of the 4-bit adder after initial refinement are:
$P_x = \{\{A_3, B_3\}, \{A_2, B_2\}, \{A_1, B_1\}, \{A_0, B_0, C_{in}\}\}$
$P_z = \{\{S_0\}, \{S_1\}, \{S_2\}, \{S_3, S_4\}\}$

4.3 Refining Outputs by Minterm Count

Lemma 3. *Two outputs can match only if their Boolean functions have the same number of minterms.*

Ordering criterion 2. Let i and j be two outputs in the same output cluster and let $M(i)$ and $M(j)$ be the number of minterms of i and j, respectively. If $M(i) < M(j)$, then the subcluster containing i is smaller than the subcluster containing j.

Minterm count is another effective output signature which is only practical when the circuit is represented in BDD form. In fact, the widely adopted way to count the minterms of a Boolean network represented in AIG is to first convert it to a BDD, but this approach is limited in scalability [16].

4.4 Refining I/O by Unateness

Lemma 4. *Two outputs match only if they are unate in the same number of inputs. Similarly, two inputs match only if the same number of outputs is unate in them.*

Ordering criterion 3. Let i and j be two outputs in the same output cluster. Assume that $Unate(i)$ and $Unate(j)$ are the number of unate variables of i and j respectively, and let $Unate(i) < Unate(j)$. Then, the output subcluster containing i is smaller than the subcluster containing j. Similarly, let i and j be two inputs in one input cluster. Assume that $Unate(i)$ and $Unate(j)$ are the number of outputs that are unate in i and j respectively, and let $Unate(i) < Unate(j)$. Then, the input subcluster containing i is smaller than the subcluster containing j.

Although unateness generates powerful signatures for Boolean matching, computing unateness in an AIG encounters the same limitation as was discussed for counting the number of minterms. Hence, refinement based on unateness is only practical for small Boolean networks.

4.5 Scalable I/O Refinement by Dependency Analysis

We mentioned earlier that the degree of each I/O is an effective signature for initial refinement of I/O sets. Here, we generalize this concept by not only considering the number of support variables but also carefully analyzing I/O dependencies.

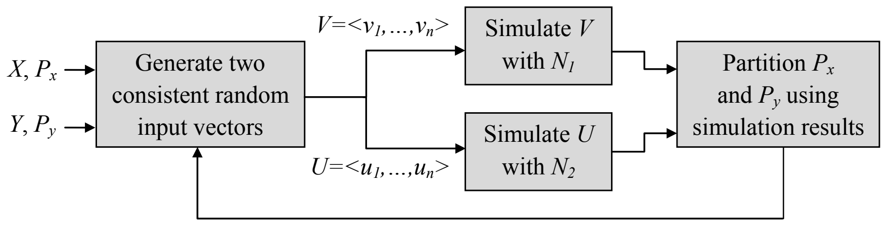
Definition 9. Let x be an input (output) and let $Supp(x) = \{z_1, \dots, z_k\}$. We define a sequence $S = (s_1, \dots, s_k)$ of unsigned integers where each s_i is the index of the output (input) cluster that z_i belongs to. After sorting S, we call it *support signature* of x and we denote it by $Sign(x)$.

Lemma 5. *Two I/Os i and j in the same I/O cluster are distinguishable if $Sign(i) \neq Sign(j)$.*

Ordering criterion 4. Let i and j be two I/Os in the same I/O cluster. Assume that $Sign(i) < Sign(j)$ meaning that the support signature of i is lexicographically smaller than the support signature of j. Then, the subcluster containing i precedes the subcluster containing j.

Example 3. Consider a circuit with input set $X = \{x_1, x_2, x_3\}$ and output set $Z = \{z_1, z_2, z_3\}$ where $z_1 = \overline{x_1}$, $z_2 = x_1 \cdot x_2$ and $z_3 = \overline{x_2 \cdot x_3}$. The I/O supports of the circuit are: $Supp(z_1) = \{x_1\}$, $Supp(z_2) = \{x_1, x_2\}$, $Supp(z_3) = \{x_2, x_3\}$ and $Supp(x_1) = \{z_1, z_2\}$, $Supp(x_2) = \{z_2, z_3\}$, $Supp(x_3) = \{z_3\}$. Since $D(z_1) = 1$ and $D(z_2) = D(z_3) = 2$, and $D(x_3) = 1$ and $D(x_1) = D(x_2) = 2$, we can initialize I/O clusters as follows: $P_z = \{\{z_1\}, \{z_2, z_3\}\}$, $P_x = \{\{x_3\}, \{x_1, x_2\}\}$. Now, we try refining based on support signatures. The signatures for z_1, z_3, x_1 and x_2 are: $Sign(z_2) = (2,2)$, $Sign(z_3) = (1,2)$, $Sign(x_1) = (1,2)$, $Sign(x_2) = (2,2)$. Since $Sign(z_3) < Sign(z_2)$ and $Sign(x_1) < Sign(x_2)$, we can further partition $\{z_2, z_3\}$ and $\{x_1, x_2\}$, hence $P_z = \{\{z_1\}, \{z_3\}, \{z_2\}\}$ and $P_x = \{\{x_3\}, \{x_1\}, \{x_2\}\}$.

After each round of refinement based on I/O dependencies, we check if any I/O cluster is further partitioned. If a new partition is added, the algorithm performs another round of refinement. The procedure terminates when no new refinement occurs after a certain number of iterations.



X, Px
Y, Py
Generate two consistent random input vectors
V=<v1,…,vn>
U=<u1,…,un>
Simulate V with N1
Simulate U with N2
Partition Px and Py using simulation results

Fig. 4 Flow of the proposed I/O refinement by random simulation.

4.6 Scalable I/O Refinement by Random Simulation

Functional simulation holds the promise to quickly prune away unpromising branches of search, but this seems to require a matching of outputs. Instead, we find pairs of input vectors that sensitize comparable functional properties of the two circuits. Let $V =< v_1,...,v_n >$ be an input vector of Boolean network N. The result of simulating V on N is called the *output vector* of N under V and is denoted by $R_v =< r_1,...,r_m >$.

Definition 10. Let N be a Boolean network with input set X and let $P_x = \{X_1,...,X_k\}$ be an ordered partition of X. An input vector $V =< v_1,...,v_n >$ is said to be *proper* if it assigns the same value (0 or 1) to all the inputs of N which are in the same input cluster, i.e., $v_i = v_j$ if $i, j \in X_l$ for some l. The input vectors consisting of all 0s or all 1s are the *trivial* proper input vectors.

Definition 11. Let $X = \{x_1,...,x_n\}$ and $Y = \{y_1,...,y_n\}$ be the input sets of two Boolean networks N_1 and N_2 and let $P_x = \{X_1,...,X_k\}$ and $P_y = \{Y_1,...,Y_k\}$ be two ordered partitions defined on them. Two proper random input vectors $V =< v_1,...,v_n >$ and $U =< u_1,...,u_n >$ of N_1 and N_2 are said to be *consistent* if, for all $1 \leq l \leq k$, $x_i \in X_l$ and $y_j \in Y_l$ imply that $v_i = u_j$.

Intuitively, two consistent random input vectors try to assign the same value to all potentially matchable inputs of the two Boolean networks. In the next three subsections, we distinguish three types of simulation based on pairs of consistent random input vectors that help us sensitize certain functional properties of the two circuits. The flow of the I/O refinement by random simulation is shown in Figure 4.

4.6.1 Simulation type 1

Lemma 6. *Let V be a proper random input vector and let $R_v =< r_1,...,r_m >$ be the corresponding output vector under V. Two outputs i and j in one output cluster are distinguishable if $r_i \neq r_j$.*

The above lemma classifies outputs based on their values (0 or 1) using the following ordering criterion.

Ordering criterion 5. The output subcluster of all 0s precedes the output subcluster of all 1s.

4.6.2 Simulation type 2

Definition 12. Let V be a proper random input vector and let $R_v =< r_1,...,r_m >$ be the corresponding output vector under V. Let V' be another input vector created by flipping the value of input x in V and let $R_{V'} =< r'_1,...,r'_m >$ be the corresponding output vector under V'. The *observability* of input x with respect to V denoted by $Obs(x)$ is defined as the number of flips in the outputs caused by V', i.e., the number of times $r_i \neq r'_i$.

Lemma 7. *Two inputs i and j in one input cluster are distinguishable if $Obs(i) \neq Obs(j)$.*

Ordering criterion 6. Let i and j be two inputs in one input cluster and let $Obs(i) < Obs(j)$. Then, the input subcluster containing i precedes the input subcluster containing j.

4.6.3 Simulation type 3

Definition 13. Consider a proper random input vector V and its corresponding output vector $R_v =< r_1,...,r_m >$. Let $V_1,...,V_n$ be n input vectors where vector V_i is created by flipping the value of r_i in V. Also, let $R_{v_1} =< r_{1,1},...,r_{1,m} >,...,R_{v_n} =< r_{n,1},...,r_{,m} >$ be the corresponding output vectors under $V_1,...,V_n$. The *controllability* of output z with respect to V denoted by $Ctrl(z)$ is defined as the number of times $r_i \neq r_{j,i}$, for all $1 \leq j \leq n$.

Lemma 8. *Two outputs i and j in one output cluster are distinguishable if $Ctrl(i) \neq Ctrl(j)$.*

Ordering criterion 7. Let i and j be two outputs in one output cluster and let $Ctrl(i) < Ctrl(j)$. Then, the output subcluster containing i precedes the output subcluster containing j.

Example 4. Consider an 8-to-1 multiplexer with input set $X = \{a_0,...,a_7,s_0,s_1,s_2\}$ and output z where $a_0,...,a_7$ denote data signals and s_0,s_1,s_2 are control signals. Initially P_x has only one partition, namely X. Initial refinement and refinement by dependency analysis do not partition P_x, hence we try random simulation. Here, we can only use type 2 simulation since simulation of type 1 and 3 are for refining output clusters. First, we consider the trivial input vector V of all 0s. We flip one input in V at a time and we apply the resulting vectors to the multiplexer. Only flipping a_0 flips z; hence, $P_x = \{\{a_1,...,a_7,s_0,s_1,s_2\},\{a_0\}\}$. Then we try the trivial input vector V of all 1s. This time flipping a_7 flips z; hence, $P_x = \{\{a_1,...,a_6,s_0,s_1,s_2\},\{a_7\},\{a_0\}\}$. Next, we put a_0 to 1 and all the other inputs to 0. Now flipping s_0,s_1,s_2 flips z, hence $P_x = \{\{a_1,...,a_6\},\{s_0,s_1,s_2\},\{a_7\},\{a_0\}\}$. If we continue partitioning based on the remaining proper input vectors no additional refinement will be gained.

After matching I/Os using random simulation, we check if any progress is achieved in refining I/O clusters. If a new cluster is added, the algorithm continues refining based on random simulation. The procedure terminates when no new refinement occurs in input or output subclusters after a certain number of iterations. Here, the number of iterations does not affect the correctness of the algorithm. However, too few iterations might diminish the impact of matching by random simulation, and excessive iterations offer no improvement. Our current implementation limits iterations to 200.

5 SAT-based Search

The scalable methods we introduced so far typically reduce the number of possible matches from $n!m!$ to hundreds or less, often making exhaustive search (with SAT-based equivalence-checking) practical. However, this phase of Boolean matching can be significantly improved, and the techniques we develop facilitate scaling to even larger instances.

5.1 SAT-based Input Matching

The basic idea in our SAT-based matching approach is to build a tree data structure called *SAT-tree* that matches one input at a time from the remaining non-singleton input clusters. Subsequently, after an input is matched, all the outputs in its support which are not matched so far are also matched, one by one. In other words, we build a dual-purpose tree that repeatedly matches inputs and outputs until exact I/O matches are found. We take advantage of the following lemma to build our SAT-tree:

Lemma 9. *Assume that two Boolean networks N_1 and N_2 with input sets $X = \{x_1,\ldots,x_n\}$ and $Y = \{y_1,\ldots,y_n\}$ are functionally equivalent under two complete ordered partitions $P_x = \{X_1,\ldots,X_n\}$ and $P_y = \{Y_1,\ldots,Y_n\}$. Also, assume that $X_l = \{x_i\}$ and $Y_l = \{y_j\}$. Let N_1' be the positive (negative) cofactor of N_1 with respect to x_i and N_2' be the positive (negative) cofactor of N_2 with respect to y_j. N_1' and N_2' with input sets $X' = X - \{x_i\}$ and $Y' = Y - \{y_j\}$ behave functionally the same under two complete ordered partitions $P_{x'} = P_x - \{X_l\}$ and $P_{y'} = P_y - \{Y_l\}$.*

The inputs to the SAT-based matching algorithm are two ordered input partitions and two ordered output partitions. Here, we assume that some of the partitions are incomplete because if all partitions are complete, an exact match is already found. Without loss of generality, assume that in two ordered partitions $P_x = \{X_1,\ldots,X_k\}$ and $P_y = \{Y_1,\ldots,Y_k\}$ of sets X and Y, $X_1,\ldots,X_{l-1}$ and $Y_1,\ldots,Y_{l-1}$ are all singleton clusters and $X_l,\ldots,X_k$ and $Y_l,\ldots,Y_k$ are non-singleton clusters. Repeatedly applying Lemma 9 allows us to create two new Boolean networks N_1' and N_2' by setting all the inputs in $X_l,\ldots,X_k$ and $Y_l,\ldots,Y_k$ to either constant 0 or constant 1. In other words, we shrink input sets X to $X' = X - \{x|x \in \{X_l,\ldots,X_k\}\}$ and input set Y to $Y' = Y - \{y|y \in \{Y_l,\ldots,Y_k\}\}$ such that X' and Y' only contain the inputs that have exact match in N_1

and N_2. Note that, by definition, the ordered partitions $P'_x = P_x - \{X_l, \dots, X_k\}$ and $P'_y = P_y - \{Y_l, \dots, Y_k\}$ are complete partitions of X' and Y'. According to Lemma 9, N'_1 and N'_2 must be functionally equivalent if N_1 and N_2 are equivalent. N'_1 and N'_2 are called the *Smallest Matching Sub-circuits* (*SMS*) of N_1 and N_2.

After finding the SMS of N_1 and N_2, we try to expand X' and Y' back to X and Y by matching one input at a time. Let X_l and Y_l be the first two non-singleton input clusters of P_x and P_y and let $x_i \in X_l$. The goal here is to match x_i with one of the $|Y_l|$ inputs in Y_l. Assume that $y_j \in Y_l$, and we pick y_j as the first candidate to match x_i. Now, in order to reflect our matching decision, we partition X_l and Y_l to make $\{x_i\}$ and $\{y_j\}$ two singleton clusters; hence, X_l is partitioned to $X_{l,1} = \{x_i\}$ and $X_{l,2} = X_l - \{x_i\}$ and Y_l is partitioned to $Y_{l,1} = \{y_j\}$ and $Y_{l,2} = Y_l - \{y_j\}$. Complying with our previous notation, now $X_{l,2}, \dots, X_k$ and $Y_{l,2}, \dots, Y_k$ are the new non-singleton clusters. We then build two Boolean networks N''_1 and N''_2 from N_1 and N_2 by setting all the inputs in non-singleton clusters to either constant 0 or constant 1, and we pass the miter of N''_1 and N''_2 to the SAT-solver. The SAT-solver may return either *satisfiable* or *unsatisfiable*. If the result is:

- ***unsatisfiable***: N''_1 and N''_2 are functionally equivalent. In other words, x_i and y_j has been a valid match so far. Hence, first try to match the outputs in the supports of x_i and y_j (only the outputs that have not been matched so far) and then match the next two unmatched inputs in $X_{l,2}$ and $Y_{l,2}$.
- ***satisfiable***: N''_1 and N''_2 are not functionally equivalent. In other words, x_i cannot match y_j. Backtrack one level up and use the counterexample to prune the SAT-tree.

In a case where the SAT-solver times out, we terminate the matching process, and only report the I/Os matched by our signature-based techniques. Unlike early prototypes, our most optimized implementation does not experience this situation on the testcases used in our experiments.

5.2 Pruning Invalid Input Matches By SAT Counterexamples

Pruning the SAT-tree using counterexamples produced by SAT is a key step in our Boolean matching methodology. Continuing the scenario in Section 5.1, assume that the miter of N''_1 and N''_2 is *satisfiable*. Suppose that the SAT-solver returns an input vector $V =< v_1, \dots, v_{l+1} >$ as the satisfying assignment. This input vector carries a crucial piece of information: the matching attempt before matching x_i and y_j was a successful match; otherwise we would have backtracked in the previous level and we would have never tried matching x_i and y_j. Thus, the input vector V sensitizes a path from x_i and y_j to the outputs of the miter.

According to Lemma 9, repeatedly computing negative and positive cofactors of N_1 and N_2 with respect to the values of $v_1, \dots, v_l$ in V results in two new Boolean networks $\hat{N}_1$ and $\hat{N}_2$ that must be functionally equivalent under some ordered partition $P_x - \{X_1, \dots, X_l\}$ and $P_y - \{Y_1, \dots, Y_l\}$. In other words, $\hat{N}_1$ and $\hat{N}_2$ are two smaller

Boolean networks that only contain the inputs of N_1 and N_2 that have not found exact match so far. Since $\hat{N}_1$ and $\hat{N}_2$ are computed with respect to the values of $v_1,\dots,v_l$ in V and since V is a vector that sensitizes a path form x_i and y_j to the output of the miter, we conclude that there exists an output in $\hat{N}_1$ that is functionally dependent on x_i. The existence of such an output ensures that $D(x_i) > 0$. We can now apply our simple filtering signature from Lemma 2 to prune the SAT-tree. Specifically, $x_i \in X_l$ can match to $y_q \in Y_l$ $(q \neq j)$ only if $D(x_i) = D(y_q)$ in $\hat{N}_1$ and $\hat{N}_2$.

Example 5. Consider two 8-to-1 multiplexers with outputs z and z' and input sets $X = \{a_0,\dots,a_7,s_0,s_1,s_2\}$ and $X' = \{a'_0,\dots,a'_7,s'_0,s'_1,s'_2\}$. Refining X and X' based on the techniques explained in Section 4 would result in two ordered partitions $P_x = \{\{a_1,\dots,a_6\},\{s_0,s_1,s_2\},\{a_7\},\{a_0\}\}$ and $P_{x'} = \{\{a'_1,\dots,a'_6\},\{s'_0,s'_1,s'_2\},\{a'_7\},\{a'_0\}\}$ (refer to Example 4). In order to find exact input matches, we build our SAT-tree and we first try matching s_2 and s'_0. The SAT-solver confirms the validity of this match. Then, s_1 matches s'_1 and s_0 matches s'_2. These two matches are also valid. So far, $P_x = \{\{a_1,\dots,a_6\},\{s_2\},\{s_1\},\{s_0\},\{a_7\},\{a_0\}\}$ and $P_{x'} = \{\{a'_1,\dots,a'_6\},\{s'_0\},\{s'_1\},\{s'_2\},\{a'_7\},\{a'_0\}\}$. Now, we look at the next non-singleton input cluster and we match a_1 and a'_1. Our SAT-solver specifies that matching a_1 and a'_1 do not form a valid match and it returns vector V in which $s'_0 = s_2 = 0$, $s'_1 = s_1 = 0$, $s'_2 = s_0 = 1$, $a'_7 = a_7 = 0$, $a'_0 = a_0 = 0$, $a'_1 = a_1 = 1$ as a counterexample. In order to see why V is a counterexample of matching a_1 and a'_1, we look at the cofactors of the two multiplexers, c and c', where all the inputs in non-singleton clusters are set to 0: $c = a_0\overline{s_2}\,\overline{s_1}\,\overline{s_0} + a_1\overline{s_2}\,\overline{s_1}s_0 + a_7 s_2 s_1 s_0$ and $c' = a'_0 s'_0 s'_1 s'_2 + a'_1 s'_0 s'_1 s'_2 + a'_7 s'_0 s'_1 s'_2$. Applying V to c and c' would result in $c = 1$ and $c' = 0$. Since we know that a_1 does not match a'_1, we use the counterexample to prune the SAT-tree. Specifically, we compute cofactors of the two multiplexers, d and d', with respect to the values of matched inputs in V. So, $d = a_1\overline{s_2}\,\overline{s_1}s_0$ and $d' = a'_4 s'_0 s'_1 s'_2$. In d and d', $D(a_1) = D(a'_4) = 1$. This means that a_1 can only match a'_4. In other words, we have pruned SAT search space by not matching a_1 to any of inputs a'_2, a'_3, a'_5 and a'_6. We continue matching inputs until we find valid matches.

5.3 SAT-based Output Matching

Let Z and W be the output sets of two Boolean networks N_1 and N_2 and let $P_z = \{Z_1,\dots,Z_k\}$ and $P_w = \{W_1,\dots,W_k\}$ be two ordered partitions defined on them. Continuing the scenario in Section 5.1, assume that $z_i \in Z_l$ is a support variable of x_i, $w_j \in W_l$ is a support variable of y_j, and Z_l and W_l are two non-singleton output clusters of P_z and P_w. In order to verify if z_i and w_j match under current input correspondence, we add $z_i \bigoplus w_j$ to the current miter of N''_1 and N''_2 and we call SAT-solver once again. If SAT returns *unsatisfiable*, i.e., z_i matches w_j, we continue matching the remaining unmatched outputs in the support of x_i and y_j. If the result is *satisfiable*, we once again use the counterexample returned by SAT to prune the search space.

Example 6. Consider two circuits N_1 and N_2 with input sets $X = \{x_0, \ldots, x_3\}$ and $Y = \{y_0, \ldots, y_3\}$, and output sets $Z = \{z_0, z_1\}$ and $W = \{w_0, w_1\}$ where $z_0 = x_0 \cdot x_1 \cdot \overline{x_2} \cdot \overline{x_3}$, $z_1 = \overline{x_0} \cdot \overline{x_1} \cdot x_2 \cdot x_3$, $w_0 = \overline{y_0} \cdot \overline{y_1} \cdot y_2 \cdot y_3$, and $w_1 = y_0 \cdot y_1 \cdot \overline{y_2} \cdot \overline{y_3}$. For these two circuits, signature-based matching (discussed in Section 4) cannot distinguish any I/Os. Hence, we resort to SAT-solving. Assume that SAT search starts by matching x_0 to y_0. Since $\{z_0, z_1\} \in Supp(x_0)$ and $\{w_0, w_1\} \in Supp(y_0)$, the outputs of the circuits must be matched next. Among all valid matches, our SAT-solver can match z_0 to w_1 and z_1 to w_0. For the remaining space of the unmatched inputs, our SAT-solver can validly match x_1 to y_1, x_2 to y_3, and x_3 to y_2, and finish the search.

5.4 Pruning Invalid Output Matches By SAT Counterexamples

When output $z_i \in Z_l$ does not match output $w_j \in W_l$, the counterexample returned by SAT is a vector V that makes $z_i = 1$ and $w_j = 0$ or vice versa. This means that z_i matches output $w_q \in W_l$ $(q \neq j)$ only if $z_i = w_q$ under V. This simple fact allows us to drastically prune our SAT-tree whenever an invalid output match occurs.

5.5 Pruning Invalid I/O Matches Using Support Signatures

We demonstrated in Section 4.5 that support signatures of inputs and outputs can be used to refine I/O subclusters of a Boolean network. In this section, we show that support signatures can also be used in our SAT-tree to eliminate impossible I/O correspondences.

Lemma 10. *Suppose that $x_i \in X_l$ and $y_j \in Y_l$ are two unmatched inputs of N_1 and N_2. Then, x_i can match y_j only if $Sign(x_i) = Sign(y_j)$. Likewise, suppose that $z_i \in Z_l$ and $w_j \in W_l$ are two unmatched outputs of N_1 and N_2. Then, z_i matches w_j only if $Sign(z_i) = Sign(w_j)$.*

As indicated in Section 5.1, matching two I/Os during SAT search introduces new singleton cells. These new cells might change the support signature of the remaining unmatched I/Os (the ones in the supports of the recently matched inputs or outputs). According to Lemma 10, this change in the support signatures might preclude some I/Os from matching. Taking advantage of this lemma, we can prune the unpromising branches of the SAT tree in the remaining space of matches.

5.6 Pruning Invalid Input Matches Using Symmetries

Our SAT-tree can exploit the symmetries of inputs to prune 1) impossible output matches, and 2) symmetric portions of the search space. Since computing the input symmetries of a Boolean network is expensive, the techniques explained in this section may in some cases hamper the matching process.

Definition 14. Let $X = \{x_1, \ldots, x_n\}$ be an input set of a Boolean network N. Let $x_i \sim x_j$ (read x_i is *symmetric* to x_j) if and only if the functionality of N stays invariant under an exchange of x_i and x_j. This defines an equivalence relation on set X, i.e., $\sim$ partitions X into a number of *symmetry classes* where each symmetry class contains all the inputs that are symmetric to each other. The partition resulting from $\sim$ is called the *symmetry partition* of X.

For multi-output functions, symmetries of inputs are reported independently for each output. In other words, each output defines its own symmetry partition on inputs. Complying with the notion of symmetry in Definition 14, for a multi-output function, x_i is called symmetric to x_j if 1) x_i and x_j have the same output support, i.e., $Supp(x_i) = Supp(x_j)$, and 2) x_i and x_j are symmetric in all the outputs in their support, i.e., $x_i \sim x_j$ for all outputs in $Supp(x_i)$ (or equivalently $Supp(x_j)$).

Symmetries of inputs can serve as a signature for matching outputs in our SAT-based search. The following lemma explains the role of symmetries in detecting invalid output matches.

Lemma 11. *Output $z_i \in Z_l$ (from N_1) matches output $w_j \in W_l$ (from N_2) only if symmetry partition of z_i is isomorphic to the symmetry partition of w_j for at least one ordering of symmetry classes.*

Input symmetries can also be used to prune symmetric parts of the search space during SAT-based exploration. Specifically, assume that the miter of N_1'' and N_2'' from Section 5.1 is *satisfiable*, i.e., x_i does not match y_j. Based on the notion of input symmetries, if x_i does not match y_j, neither can it match another input in Y_l that is symmetric to y_j. In other words, x_i cannot match $y_q \in Y_l$, if y_j and y_q are symmetric.

In practice, the use of symmetries in Boolean matching encounters two major limitations: 1) finding symmetries of a large Boolean network usually takes a significant amount of time, 2) in a case where a Boolean network does not have much symmetry, a considerable amount of time can be wasted.

5.7 A Heuristic for Matching Candidates

In order to reduce the branching factor of our SAT-tree, we first match I/Os of smaller I/O clusters. Also, within one I/O cluster, we exploit the observability of the inputs and the controllability of the outputs, to make more accurate guesses in our SAT-based matching approach. Heuristically, the probability that two I/Os match is higher when their observability/controllability are similar. We observed that, in many designs, the observability of control signals is higher than that of data signals. Therefore, we first match control signals. This simple heuristic can greatly improve the runtime — experiments indicate that once control signals are matched, data signals can be matched quickly.

6 Empirical Validation

We have implemented the proposed approach in ABC and we have experimentally evaluated its performance on a 2.67GHz Intel Xeon CPU running Windows Vista. Table 1 and Table 2 show the runtime of our algorithms on ITC'99 benchmarks for P-equivalence and PP-equivalence checking problems, respectively. In these two tables, #I is the number of inputs, #O is the number of outputs and $|AIG|$ is the number of nodes in the AIG of each circuit. The last four columns demonstrate the initialization time (computing I/O support variables, initially refining I/O cluster and refining based on I/O dependencies), simulation time, SAT time, and overall time for each testcase. In addition to the reported runtimes, (I%) and (I%,O%) show the percentage of inputs and I/Os that are matched after each step. Note that, in these experiments, we did not perform refinement using minterm counts and unateness, and we did not account for input symmetries to prune our SAT-tree because these techniques appear less scalable than the ones reported in Table 1 and Table 2. Also note that for each testcase we generated 20 new circuits each falling into one of the two following categories: (1) permuting inputs for verifying P-equivalence (2) permuting both inputs and outputs for verifying PP-equivalence. The results given in Table 1 and Table 2 are the average results over all the generated testcases for each category. Furthermore, the AIGs of the new circuits are reconstructed using ABC's combinational synthesis commands to ensure that the new circuits are structurally different from the original ones.

Table 1 P-equivalence runtime (sec.) and percentage of matched inputs for ITC'99 benchmarks

Circuit	#I	#O	$\|AIG\|$	Initialization	Simulation	SAT	Overall
b01	6	7	48	**0.30** (66%)	**0** (100%)	**0** (100%)	**0.30**
b02	4	5	28	**0.28** (50%)	**0** (100%)	**0** (100%)	**0.28**
b03	33	34	157	**0.36** (97%)	**0** (97%)	**0.04** (100%)	**0.40**
b04	76	74	727	**0.41** (64%)	**0.04** (100%)	**0** (100%)	**0.45**
b05	34	70	998	**0.52** (84%)	**0.02** (100%)	**0** (100%)	**0.54**
b06	10	15	55	**0.37** (80%)	**0** (100%)	**0** (100%)	**0.37**
b07	49	57	441	**0.41** (67%)	**0.01** (100%)	**0** (100%)	**0.43**
b08	29	25	175	**0.36** (90%)	**0** (100%)	**0** (100%)	**0.36**
b09	28	29	170	**0.40** (100%)	**0** (100%)	**0** (100%)	**0.40**
b10	27	23	196	**0.34** (85%)	**0** (100%)	**0** (100%)	**0.34**
b11	37	37	764	**0.40** (95%)	**0.01** (100%)	**0** (100%)	**0.41**
b12	125	127	1072	**0.38** (60%)	**0.25** (100%)	**0** (100%)	**0.63**
b13	62	63	353	**0.38** (71%)	**0.01** (100%)	**0** (100%)	**0.39**
b14	276	299	10067	**6.89** (73%)	**3.29** (100%)	**0** (100%)	**10.18**
b15	484	519	8887	**14.26** (57%)	**5.82** (100%)	**0** (100%)	**20.08**
b17	1451	1512	32290	**246** (63%)	**46.14** (99%)	**1.41** (100%)	**294**
b18	3357	3343	74900	**2840** (69%)	**51.6** (99%)	**2.96** (100%)	**2895**
b20	521	512	20195	**52.8** (83%)	**2.23** (100%)	**0.01** (100%)	**55**
b21	521	512	20540	**52.8** (83%)	**2.30** (100%)	**0.01** (100%)	**55**
b22	766	757	29920	**150** (82%)	**3.85** (100%)	**0.32** (100%)	**154**

Table 2 PP-equivalence runtime (sec.) and percentage of matched I/Os for ITC'99 benchmarks

Circuit	#I	#O	|AIG|	Initialization	Simulation	SAT	Overall
b01	6	7	48	**0.37** (50%, 43%)	**0** (83%, 85%)	**0.02** (100%, 100%)	**0.39**
b02	4	5	28	**0.28** (50%, 60%)	**0** (100%, 100%)	**0** (100%, 100%)	**0.28**
b03	33	34	157	**0.38** (48%, 38%)	**0.01** (54%, 47%)	**0.43** (100%, 100%)	**0.82**
b04	76	74	727	**0.37** (16%, 13%)	**0.1** (100%, 100%)	**0** (100%, 100%)	**0.47**
b05	34	70	998	**0.51** (34%, 24%)	**0.03** (54%, 47%)	**0.33** (100%, 100%)	**0.87**
b06	10	15	55	**0.39** (30%, 47%)	**0** (50%, 53%)	**0.04** (100%, 100%)	**0.43**
b07	49	57	441	**0.43** (67%, 70%)	**0.03** (94%, 95%)	**0.19** (100%, 100%)	**0.65**
b08	29	25	175	**0.41** (27%, 36%)	**0.12** (100%, 100%)	**0** (100%, 100%)	**0.53**
b09	28	29	170	**0.41** (46%, 48%)	**0.01** (46%, 48%)	**0.20** (100%, 100%)	**0.62**
b10	27	23	196	**0.37** (88%, 95%)	**0** (100%, 100%)	**0** (100%, 100%)	**0.37**
b11	37	37	764	**0.41** (65%, 65%)	**0** (100%, 100%)	**0.02** (100%, 100%)	**0.43**
b12	125	127	1072	**0.38** (21%, 25%)	**1.05** (41%, 41%)	—	—
b13	62	63	353	**0.35** (43%, 50%)	**0.05** (97%, 97%)	**0.14** (100%, 100%)	**0.54**
b14	276	299	10067	**7.99** (72%, 58%)	**3.89** (89%, 90%)	**27** (100%, 100%)	**38.8**
b15	484	519	8887	**16.40** (62%, 67%)	**45.6** (94%94, %)	**6.30** (100%, 100%)	**68.3**
b17	1451	1512	32290	**249** (62%, 65%)	**229** (94%, 94%)	**148** (100%, 100%)	**626**
b18	3357	3343	74900	**2862** (65%, 63%)	**530** (93%, 93%)	—	—
b20	521	512	20195	**53.3** (70%, 51%)	**13.82** (89%, 89%)	**146** (100%, 100%)	**213**
b21	521	512	20540	**53.3** (70%, 51%)	**11.70** (89%, 89%)	**159** (100%, 100%)	**225**
b22	766	757	29920	**151** (70%, 50%)	**26.28** (88%, 88%)	**473** (100%, 100%)	**650**

— indicates runtime > 5000 sec.

In the ITC'99 benchmark suite, 18 circuits out of 20 have less than a thousand I/Os. Checking P-equivalence and PP-equivalence for 12 out of these 18 circuits takes less than a second. There is only one circuit (b12) for which our software cannot match I/Os in 5000 seconds. The reason is that, for b12, 1033 out of 7750 input pairs (13%) are symmetric and since our implementation does not yet account for symmetries, our SAT-tree repeatedly searches symmetric branches that do not yield valid I/O matches. For b20, b21 and b22 and for b17 and b18 with more than a thousand I/Os, computing functional dependency is the bottleneck of the overall matching runtime. Note that checking PP-equivalence for b18 results in a very large SAT-tree that cannot be resolved within 5000 seconds, although our refinement techniques before invoking SAT find exact matches for 3123 out of 3357 inputs (93%) and 3111 out of 3343 outputs (93%).

The results in Table 1 and Table 2 assume that target circuits are equivalent. In contrast, Table 3 considers cases where input circuits produce different output values on at least some inputs. For this set of experiments, we construced 20 inequivalent circuits for each testcase, using one of the following rules:

1. **Wrong signals**: outputs of two random gates were swapped.
2. **Wrong polarity**: an inverter was randomly added or removed.
3. **Wrong gate**: functionality of one random gate was altered.

In Table 3, columns Init, Sim, and SAT demonstrate the number of testcases (out of 20) for which our algorithms were able to prove inequivalence during initialization, simulation, and SAT search phases, respectively. Also, column Time shows

Table 3 P-equivalence and PP-equivalence runtime (sec.) for ITC'99 benchmarks when mismatch exists

Circuit	#I	#O	|AIG|	P-equivalence				PP-equivalence			
				Init	Sim	SAT	Time	Init	Sim	SAT	Time
b01	6	7	48	4	2	14	**0.30**	1	13	6	**0.49**
b02	4	5	28	0	10	10	**0.27**	2	12	6	**0.33**
b03	33	34	157	9	0	11	**0.35**	10	7	3	**0.45**
b04	76	74	727	8	2	10	**0.42**	13	4	3	**0.39**
b05	34	70	998	7	0	13	**0.53**	6	10	4	**0.70**
b06	10	15	55	3	3	14	**0.31**	14	5	1	**0.46**
b07	49	57	441	10	0	10	**0.43**	15	1	4	**0.71**
b08	29	25	175	9	2	9	**0.36**	12	6	2	**0.46**
b09	28	29	170	4	1	15	**0.40**	10	4	6	**0.45**
b10	27	23	196	10	5	5	**0.33**	11	3	6	**0.31**
b11	37	37	764	5	0	15	**0.40**	10	2	8	**0.53**
b12	125	127	1072	6	10	4	**0.45**	10	8	2	**3.5**
b13	62	63	353	6	9	5	**0.38**	7	7	6	**0.55**
b14	276	299	10067	3	0	17	**9.89**	10	3	7	**10.65**
b15	484	519	8887	4	2	14	**20.03**	8	4	8	**38.2**
b17	1451	1512	32290	11	0	9	**260**	3	7	10	**373**
b18	3357	3343	74900	2	0	18	**2864**	0	9	11	—[a]
b20	521	512	20195	7	0	13	**54**	1	4	15	**75.4**
b21	521	512	20540	2	0	18	**54**	5	11	4	**59.4**
b22	766	757	29920	7	1	12	**154**	0	4	16	**181**

— indicates runtime $>$ 5000 sec.
[a] The average runtime excluding instances requiring SAT was 2902 sec.

the average runtime of our matcher for the P-equivalence and PP-equivalence problems. According to the results, our matcher resorts to SAT-solving in 45% of the testcases which suggests that many of our instances are not particularly easy. Moreover, calling SAT is due to the fact that our mismatched instances were all generated with minimal changes to the original circuits. Note that, even in the case of a slight mismatch, our signature-based techniques alone could effectively discover inequivalence for 55% of testcases. Furthermore, comparing the results in Table 2 and Table 3, PP-equivalence checking is up to 4 times faster when mismatch exists. In particular, for b12, our matcher could confirm inequivalence in less than 5 seconds, even when SAT-solving was invoked. The reason is that in the case of a mismatch, our SAT-tree usually encounters invalid I/O matches early in the tree, which results in a vast pruning in the space of invalid matches.

In order to compare our work to that in [17], we have tested our algorithms on circuits from [17] that have more than 150 inputs. Results are listed in Table 4. For the results reported from [17], Orig, Unate and +Symm respectively show the runtime when no functional property is used, only functional unatness is used and, both unateness and symmetries are used. Note that experiments reported in [17] used 3GHz Intel CPUs, while our runs were on a 2.67GHz Intel CPU. To make the numerical comparisons entirely fair, our runtimes would need to be multiplied by 0.89. However, we omit this step, since our raw runtimes are already superior in

Table 4 P-equivalence runtime (sec.) compared to runtime (sec.) from [17]

Circuit	#I	#O	P-equivalence Runtime (sec.)				CPU Time (sec.) in [17]		
			Init	Sim	SAT	Overall	Orig	+Unate	+Sym
C2670	233	140	0.14	1.18	—	—	—	—	**7.96**
C5315	178	123	0.33	0.11	0.06	**0.5**	**6.31**	**2.86**	**3.29**
C7552	207	108	0.51	3.76	4.83	**9.10**	—	—	**14.56**
des	256	245	0.38	0.07	0	**0.45**	**10.21**	**0.25**	**2.33**
i10	257	224	0.43	1.03	1.23	**2.69**	**25.63**	**15.16**	**17.56**
i2	201	1	0.34	0.28	—	—	—	—	**1.02**
i4	192	6	0.31	0.27	—	—	—	—	**0.22**
i7	199	67	0.36	0.18	0	**0.54**	**0.82**	**0.04**	**0.19**
pair	173	137	0.32	0.14	0	**0.46**	**0.84**	**0.64**	**2.44**
s3384	226	209	0.10	0.25	0.47	**0.82**	**4.79**	**2.14**	**4.02**
s5378	199	213	0.11	0.53	0.63	**1.27**	**1.31**	**3.38**	**2.42**
s9234	247	250	3.11	0.53	2.85	**6.49**	**3.41**	**5.84**	**7.82**
s38584	1464	1730	58	1.66	1.54	**61**	**76**	**210**	**458**
s38417	1664	1742	50	9.46	30.9	**90**	**91**	**324**	**999**

— indicates runtime $>$ 5000 sec.

many cases. According to Table 4, our matching algorithm times out in 5000 seconds on C2670, i2 and i4. This is again due to the symmetries that are present in the inputs of these circuits. Note that the approach in [17] cannot solve these three circuits without symmetry search, either. For some other circuits, such as C7552, our approach verifies P-equivalence in less than 10 seconds but the approach in [17] cannot find a match without invoking symmetry finder. It is also evident from the results that checking P-equivalence for very large circuits, such as s38584 and s38417, is 3.5-11 times slower when symmetry finding and unateness calculations are performed during Boolean matching. This confirms our intuition that symmetry and unateness are not essential to Boolean matching in many practical cases, although they may occasionally be beneficial.

7 Chapter Summary

In this chapter, we proposed techniques for solving large-scale PP-equivalence checking problem. Our approach integrates graph-based, simulation driven and SAT-based techniques to efficiently solve the problem. Graph-based techniques limit dependencies between inputs and outputs and are particularly useful with word-level arithmetic circuits. Simulation quickly discovers inputs on which inequivalent circuits differ. Equivalences are confirmed by invoking SAT, and these invocations are combined with branching on possible matches. Empirical validation of our approach on available benchmarks confirms its scalability to circuits with thousands of inputs and outputs. Future advances in Boolean matching, as well as many existing techniques, can also be incorporated into our framework to improve its scalability.

References

1. Abdollahi, A.: Signature based boolean matching in the presence of don't cares. In: DAC '08: Proceedings of the 45th annual Design Automation Conference, pp. 642–647. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1391469.1391635
2. Abdollahi, A., Pedram, M.: A new canonical form for fast boolean matching in logic synthesis and verification. In: DAC '05: Proceedings of the 42nd annual Design Automation Conference, pp. 379–384. ACM, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1065579.1065681
3. Agosta, G., Bruschi, F., Pelosi, G., Sciuto, D.: A unified approach to canonical form-based boolean matching. In: DAC '07: Proceedings of the 44th annual Design Automation Conference, pp. 841–846. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1278480.1278689
4. Benini, L., Micheli, G.D.: A survey of boolean matching techniques for library binding. ACM Transactions on Design Automation of Electronic Systems **2**, 193–226 (1997)
5. Chai, D., Kuehlmann, A.: Building a better boolean matcher and symmetry detector. In: DATE '06: Proceedings of the conference on Design, automation and test in Europe, pp. 1079–1084. European Design and Automation Association, 3001 Leuven, Belgium, Belgium (2006)
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962). DOI http://doi.acm.org/10.1145/368273.368557
7. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960). DOI http://doi.acm.org/10.1145/321033.321034
8. Eén, N., Sörensson, N.: An extensible SAT-solver. In: E. Giunchiglia, A. Tacchella (eds.) SAT, *Lecture Notes in Computer Science*, vol. 2919, pp. 502–518. Springer (2003)
9. Goering, R.: Xilinx ISE handles incremental changes. http://www.eetimes.com/showArticle.jhtml?articleID=196901122 (2007)
10. Krishnaswamy, S., Ren, H., Modi, N., Puri, R.: Deltasyn: an efficient logic difference optimizer for ECO synthesis. In: ICCAD '09: Proceedings of the 2009 International Conference on Computer-Aided Design, pp. 789–796. ACM, New York, NY, USA (2009). DOI http://doi.acm.org/10.1145/1687399.1687546
11. Lee, C.C., Jiang, J.H.R., Huang, C.Y.R., Mishchenko, A.: Scalable exploration of functional dependency by interpolation and incremental SAT solving. In: ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design, pp. 227–233. IEEE Press, Piscataway, NJ, USA (2007)
12. Mishchenko, A.: Logic synthesis and verification group. ABC: A system for sequential synthesis and verification, release 70930. http://www.eecs.berkeley.edu/ãlanmi/abc/
13. Mishchenko, A., Chatterjee, S., Brayton, R.: FRAIGs: A unifying representation for logic synthesis and verification. Tech. rep., UC Berekeley (2005)
14. Mishchenko, A., Chatterjee, S., Brayton, R., Een, N.: Improvements to combinational equivalence checking. In: ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design, pp. 836–843. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1233501.1233679
15. Ray, S., Mishchenko, A., Brayton, R.: Incremental sequential equivalence checking and subgraph isomorphism. In: Proc. of the Intl. Workshop on Logic Synthesis, pp. 37–42 (2009)
16. S. Nocco, S.Q.: A probabilistic and approximated approach to circuit-based formal verification. Journal of Satisfiability, Boolean Modeling and Computation **5**, 111–132 (2008)
17. Wang, K.H., Chan, C.M., Liu, J.C.: Simulation and SAT-based boolean matching for large boolean networks. In: DAC '09: Proceedings of the 46th Annual Design Automation Conference, pp. 396–401. ACM, New York, NY, USA (2009). DOI http://doi.acm.org/10.1145/1629911.1630016

Index