BooM: A Decision Procedure for Boolean Matching with Abstraction and Dynamic Learning

Chih-Fan Lai
GIEE
National Taiwan University
Taipei 10617, Taiwan

Jie-Hong R. Jiang
EE Dept./GIEE
National Taiwan University
Taipei 10617, Taiwan
jhjiang@cc.ee.ntu.edu.tw

Kuo-Hua Wang
CSIE Dept.
Fu Jen Catholic University
Hsinchuang 24205, Taiwan

ABSTRACT

Boolean matching determines whether two given (in)completely-specified Boolean functions can be identical or complementary to each other under permutation and/or negation of their input variables. Due to its broad applications in logic synthesis and verification, it attracted much attention. Most prior efforts however were incomplete and/or restricted to certain special matching conditions. In contrast, this paper focuses on the computation kernel of Boolean matching and proposes a complete generic framework. Through conflict-driven learning and abstraction, the capacity of Boolean matching scales up due to the effective pruning of infeasible matching solutions. Experiments show encouraging results in resolving hard instances that are otherwise unsolvable.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*automatic synthesis, verification*

General Terms

Algorithms, design, verification

Keywords

Boolean matching, satisfiability solving, learning, abstraction

1. INTRODUCTION

Boolean matching is an important subject of both theoretical and practical interest. Given two (in)completely-specified Boolean functions, Boolean matching (under NPN-equivalence) determines if they can be identical or complementary to each other under certain input permutation and/or negation. From a theoretical standpoint, the computational complexity of Boolean matching (known as Boolean congruence, Boolean isomorphism, and other variants, which received attention dating back to the nineteenth century) situates in between coNP and Σ_2^p in the polynomial hierarchy [7, 4]. Thus it is a good candidate for examining the open question about the collapse of the polynomial hierarchy. From a practical standpoint, it has broad applications in logic synthesis and verification, for instance, library binding [5], FPGA technology mapping [8], logic verification [13], engineering change order [11], etc.

The demand from practical applications drives the advances of Boolean matching algorithms over the last two decades. By the survey [5] and recent developments [3, 2, 17, 19], prior methods can be classified into four categories: spectral, signature-based, canonical-form-based, and, the more recent, SAT-based methods. Spectral methods are complete, but often with prohibitive computation cost. Signature-based methods are effective, but mostly incomplete due to their intrinsic limitations [12]. Canonical-form-based methods are complete, and have been improved recently, see, e.g., [2] for a review. SAT-based methods are complete [17, 19]; however, the strengths of modern propositional satisfiability (SAT) solvers have not been fully exploited yet. Among prior Boolean matching techniques, only a few took the general NPN-equivalence into account. Even if this general problem is considered, their computation costs may still be too expensive to be practical. Furthermore, spectral, signature-based, and canonical-form-based methods are not easily extendable to deal with incompletely specified functions, see, e.g., [18, 1] for a recent treatment.

We show that these shortcomings can be naturally resolved under a general computation framework, named BooM, a decision procedure dedicated to Boolean matching. In essence Boolean matching is a decision problem of solving quantified Boolean formulas (QBFs). Particularly its special problem structure allows effective conflict-driven learning for search space reduction, which is beyond the capability of a generic QBF solver.

The features of BooM are summarized as follows. 1) It handles NPN-equivalence, and both completely and incompletely specified functions. 2) It is equipped with abstraction and dynamic learning techniques for effective search space reduction. 3) It supports the search of one matching solution and of all solutions. 4) It admits easy integration with signature-based techniques for search space reduction, and helps signature-based Boolean matching be complete. 5) It uses memory efficient data structures, specifically, and-inverter graphs (AIGs) and conjunctive normal form (CNF) formulas, for scalable Boolean function representation. Experimental results show the effectiveness of BooM in conquering instances hard to solve without learning and abstraction.

Compared with the closest prior work [17], BooM outperforms it and can be much scalable. This prior method relies on the sum-of-products representation of the two functions to be matched, and consists of two separate computation phases: first learning and second SAT solving. Since there is no feedback between these two phases, the first phase can be interpreted as *static* learning in a sense. In contrast, our learning and SAT solving are interactive (with feedback). In-

DAC 2010, June 13-18, 2010, Anaheim, California, USA.

feasible solutions are learned from a conflict produced by SAT solving; SAT solving searches a solution in the space refined by learning. Hence the learning in BooM is more global and *dynamic*. As SOP representation may not be always available especially for large designs, the prior method can be limited.

This paper is organized as follows. Section 2 gives the preliminaries. Boolean matching under NPN-equivalence is presented in Section 3, and P-equivalence in Section 4. Experimental evaluation is provided in Section 5. Finally Section 6 concludes this paper and outlines future work.

2. PRELIMINARIES

As conventional notation, a set of Boolean variables is denoted with an upper-case letter, e.g., X; its elements are in lower-case letters, e.g., $x_i \in X$. The ordered version (namely, vector) of a set $X = \{x_1, \ldots, x_n\}$ is denoted as $\vec{x} = (x_1, \ldots, x_n)$. The cardinality of a set X (respectively $\vec{x}$) is denoted as $|X|$ (respectively $|\vec{x}|$). The set of truth valuations of $\vec{x}$ is denoted $[\![\vec{x}]\!]$, e.g., $[\![(x_1, x_2)]\!] = \{(0,0),(0,1),(1,0),(1,1)\}$.

2.1 Boolean Matching

A **permutation** π over X is a bijection function $\pi : X \rightarrow X$; a **negation** ν over X is a componentwise mapping with $\nu(x_i) = x_i$ or $\neg x_i$. We let $\pi(\vec{x})$ and $\nu(\vec{x})$ be the shorthand for $(\pi(x_1), \ldots, \pi(x_n))$ and $(\nu(x_1), \ldots, \nu(x_n))$, respectively.

Given two (completely specified) functions $f(\vec{x})$ and $g(\vec{y})$ with $|\vec{x}| = |\vec{y}|$, **Boolean matching** under **NPN-equivalence** determines if these two functions can be equivalent or complementary (the second "N" of "NPN") to each other under negation (the first "N") and permutation (the "P") of their input variables. Its special cases include **NP-equivalence** (determining if $f(\vec{x}) = g(\nu \circ \pi(\vec{x}))$ for some negation ν and permutation π) and **P-equivalence** (determining if $f(\vec{x}) = g(\pi(\vec{x}))$ for some permutation π). We call $\nu \circ \pi$ (respectively π) a **matching solution** for NP-equivalence (respectively P-equivalence) if $f(\vec{x}) = g(\nu \circ \pi(\vec{x}))$ (respectively $f(\vec{x}) = g(\pi(\vec{x}))$). In the sequel, unless otherwise said, we shall assume $|\vec{x}| = |\vec{y}| = n$.

Boolean matching for two incompletely specified functions is similar, except that the functional equivalence is asserted only under the care-conditions of both functions.

2.2 Propositional Satisfiability

By assuming the reader's familiarity with circuit-to-CNF conversion [16] and SAT solving, including conflict-based learning [15] and other commonly used techniques in modern SAT solvers, e.g., [14, 10], we omit to provide the background knowledge.

3. MATCHING FOR NPN-EQUIVALENCE

Given two functions $f(\vec{x})$ and $g(\vec{y})$, optionally with their care-conditions $f_c(\vec{x})$ and $g_c(\vec{y})$, respectively, we decide whether f and g can be NPN-equivalent under the care-conditions. In particular, this problem can be divided into two subtasks by deciding whether f and g or whether f and $\neg g$ are NP-equivalent. Consequently we focus on Boolean matching under NP-equivalence. Formally speaking, this task is to decide the validity of the second-order formula

$$\exists \nu \circ \pi, \forall \vec{x}.((f_c(\vec{x}) \wedge g_c(\nu \circ \pi(\vec{x}))) \Rightarrow (f(\vec{x}) \equiv g(\nu \circ \pi(\vec{x})))), \quad (1)$$

which is convertible to a first-order formula as shown below.

To represent $\vec{y} = \nu \circ \pi(\vec{x})$, we introduce the 0-1 matrix:

$$\begin{array}{c} \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{array} \begin{array}{c} \begin{array}{ccccccc} x_1 & \neg x_1 & x_2 & \neg x_2 & \cdots & x_n & \neg x_n \end{array} \\ \begin{pmatrix} a_{11} & b_{11} & a_{12} & b_{12} & \cdots & a_{1n} & b_{1n} \\ a_{21} & b_{21} & a_{22} & b_{22} & \cdots & a_{2n} & b_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & b_{n1} & a_{n2} & b_{n2} & \cdots & a_{nn} & b_{nn} \end{pmatrix} \end{array} \quad (2)$$

By asserting

$$\sum_{j=1}^{n}(a_{ij} + b_{ij}) = 1 \qquad \text{for } i = 1, \ldots, n, \text{ and} \quad (3)$$

$$\sum_{i=1}^{n}(a_{ij} + b_{ij}) = 1 \qquad \text{for } j = 1, \ldots, n, \quad (4)$$

this matrix represents some legal mapping $\nu \circ \pi$, for $a_{ij} = 1$ and $b_{ij} = 1$ indicating mapping x_j to y_i and mapping $\neg x_j$ to y_i, respectively. These cardinality constraints (3) and (4) can be expressed by a propositional formula φ_C of $2n^2$ Boolean variables a_{ij} and b_{ij}, for $i, j = 1, \ldots, n$. By asserting the formula

$$\varphi_A = \bigwedge_{i,j=1}^{n}((a_{ij} \Rightarrow (y_i \equiv x_j))(b_{ij} \Rightarrow (y_i \equiv \neg x_j))),$$

a solution to φ_C induces some unique mapping $\nu \circ \pi$. Conversely a mapping $\nu \circ \pi$ corresponds to some unique solution to φ_C. Hence in the sequel we shall not distinguish a mapping $\nu \circ \pi$ and its corresponding solution to φ_C. In practice, φ_C and φ_A are written in CNF.

Clearly solving Formula (1) is equivalent to solving the following (first-order) QBF

$$\exists \vec{a}, \exists \vec{b}, \forall \vec{x}, \forall \vec{y}.(\varphi_C \wedge \varphi_A \wedge ((f_c \wedge g_c) \Rightarrow (f \equiv g))), \quad (5)$$

where $\vec{a} = (a_{11}, \ldots, a_{nn})$ and $\vec{b} = (b_{11}, \ldots, b_{nn})$. That is, we look for a truth assignment to variables $\vec{a}$ and $\vec{b}$ that satisfies φ_C and makes the miter constraint

$$\Psi = \varphi_A \wedge f_c \wedge g_c \wedge (f \not\equiv g) \quad (6)$$

unsatisfiable. For simplicity, unless otherwise said we shall assume that f_c and g_c are tautologies in the sequel.

The key to solving Formula (5) is to effectively reduce the search space. By exploiting the functional properties specific to f and g (such as variable symmetry, unateness, and other functional properties), a preprocessing step is possible to screen out from φ_C a substantial amount of infeasible solutions. Let formula $\Phi^{\langle 0 \rangle}$ characterize the remaining solutions of variables $\vec{a}$ and $\vec{b}$ after the preprocessing. We show below how the solution space corresponding to legal $\nu \circ \pi$ can be effectively refined in a sequence $\Phi^{\langle 0 \rangle}$, $\Phi^{\langle 1 \rangle}$, ..., $\Phi^{\langle k \rangle}$, $(\Phi^{\langle i+1 \rangle} \Rightarrow \Phi^{\langle i \rangle})$, along the solution search process.

3.1 Boolean Matching with Dynamic Learning

We discuss two different Boolean matching goals: to search one matching solution and to search all matching solutions.

3.1.1 *Searching one matching solution*

Figure 1 sketches the procedure for finding one solution. It takes on two input functions f and g, possibly with their care-conditions f_c and g_c. A preprocessing step is first conducted to strengthen φ_C yielding $\Phi^{\langle 0 \rangle}$. It fast prunes infeasible matching solutions based on functional properties and an abstraction technique, to be explained in Section 3.2. After preprocessing, an iterative procedure is conducted between two interacting SAT solving instances to refine the solution space. The first SAT solving instance solves $\Phi^{\langle i \rangle}$, i.e., the

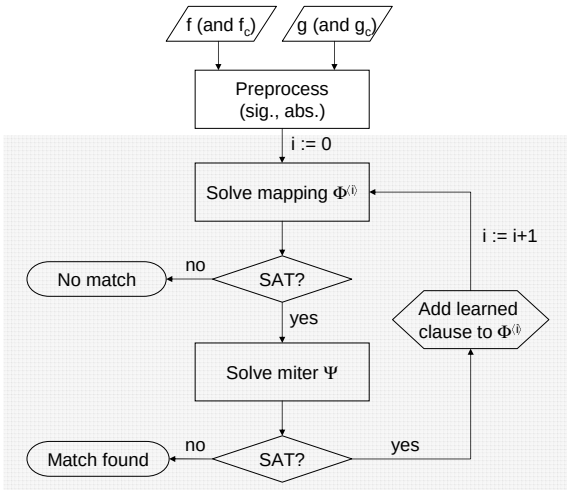

Figure 1: Search one Boolean matching solution

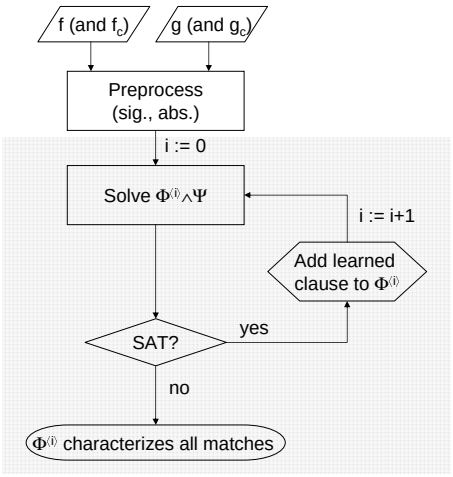

Figure 2: Search all Boolean matching solutions

remaining matching solutions after the ith iteration. No solution to $\Phi^{\langle i \rangle}$ indicates that f and g cannot be matched; otherwise, a solution to $\Phi^{\langle i \rangle}$ corresponds to a candidate mapping $\nu \circ \pi$. Further justification by the second SAT solving instance is needed to check whether this solution makes the miter formula Ψ unsatisfiable. If yes, the procedure terminates since $\nu \circ \pi$ is indeed a matching solution. Otherwise, the procedure learns from this (conflict) solution and strengthens $\Phi^{\langle i \rangle}$ to $\Phi^{\langle i+1 \rangle}$ accordingly. This action blocks not only this current solution of $\Phi^{\langle i \rangle}$ but also other infeasible solutions from occurrence in later search. (Ordinary learning blocks only the current conflict solution.) The procedure continues with $\Phi^{\langle i+1 \rangle}$ in the new iteration.

Below we show how to strengthen $\Phi^{\langle i \rangle}$ through learning.

FACT 1. *Given two functions $f(\vec{x})$ and $g(\vec{y})$ for Boolean matching under NP-equivalence, if $f(\vec{u}) \neq g(\vec{v})$ for $\vec{u} \in [\![\vec{x}]\!]$ and $\vec{v} \in [\![\vec{y}]\!]$, then any mapping $\nu \circ \pi$ with $\nu \circ \pi(\vec{u}) = \vec{v}$ is infeasible.*

EXAMPLE 1. *For two functions $f(x_1, x_2, x_3)$ and $g(y_1, y_2, y_3)$ with $f(1,0,1) \neq g(0,1,1)$, then f and g cannot be matched under NP-equivalence by the six mappings with $\vec{y} = (\neg x_1, \neg x_2, x_3)$, $(\neg x_1, x_3, \neg x_2)$, (x_2, x_1, x_3), (x_2, x_3, x_1), $(\neg x_3, x_1, \neg x_2)$, and $(\neg x_3, \neg x_2, x_1)$.*

Fact 1 can be rephrased in the language of formulas $\Phi^{\langle i \rangle}$ and Ψ as follows.

PROPOSITION 1. *Given two functions $f(\vec{x})$ and $g(\vec{y})$ for Boolean matching under NP-equivalence, if $f(\vec{u}) \neq g(\vec{v})$ for $\vec{u} \in [\![\vec{x}]\!]$ and $\vec{v} \in [\![\vec{y}]\!]$ with $\vec{v} = \nu \circ \pi(\vec{u})$ for some $\nu \circ \pi$ satisfying $\Phi^{\langle i \rangle}$, then conjuncting $\Phi^{\langle i \rangle}$ with the clause $\kappa = \bigvee_{i,j=1}^{n} l_{ij}$ for literals*

$$l_{ij} = \begin{cases} a_{ij}, & \text{if } v_i \neq u_j; \\ b_{ij}, & \text{otherwise,} \end{cases}$$

excludes from $\Phi^{\langle i \rangle}$ exactly the mappings $\{\nu' \circ \pi' \mid \nu' \circ \pi'(\vec{u}) = \nu \circ \pi(\vec{u})\}$.

In essence a satisfying solution to the miter formula Ψ with respect to a solution of $\Phi^{\langle i \rangle}$ reveals additional infeasible matching solutions. Letting $\Phi^{\langle i+1 \rangle} = \Phi^{\langle i \rangle} \wedge \kappa$ prevents the above procedure from searching the learned infeasible solutions in later iterations. As a result, a single clause with n^2 literals is added in each iteration.[1]

[1]The number of literals of a learned clause can be reduced if the underlying SAT solver is capable of providing partially assigned satisfying solutions.

EXAMPLE 2. *Consider Example 1. The learned clause corresponding to $f(1,0,1) \neq g(0,1,1)$ is*

$$(a_{11} \vee b_{12} \vee a_{13} \vee b_{21} \vee a_{22} \vee b_{23} \vee b_{31} \vee a_{32} \vee b_{33}).$$

It excludes the previously listed six infeasible mappings.

The pruning power of a learned clause can be characterized as follows.

PROPOSITION 2. *For Boolean matching under NP-equivalence, the clause κ learned from a satisfying solution to $f(\vec{x}) \not\equiv g(\vec{y})$ with $\vec{y} = \nu \circ \pi(\vec{x})$ for some $\nu \circ \pi$ prunes $n!$ infeasible mappings.*

However the sets of mappings pruned by two different learned clauses may not be disjoint.

Since there are 2^{2n} distinct truth assignments to variables $\vec{x}$ and $\vec{y}$, the number of different learned clauses is upper bounded by 2^{2n}. This fact also asserts the termination of the procedure.

PROPOSITION 3. *The Boolean matching procedure of Figure 1 for NP-equivalence terminates within $O(2^{2n})$ iterations.*

3.1.2 Searching all matching solutions

When the Boolean matching objective is to find all matching solutions rather than just one, applying the procedure of Figure 1 to find the solutions one by one can be overkill. Figure 2 sketches a more effective procedure for this purpose. This procedure is the same as that of Figure 1 except that there is only one SAT solving kernel for one combined formula $\Phi^{\langle i \rangle} \wedge \Psi$.

Unlike the procedure of Figure 1 (where, effectively, variables $\vec{a}$ and $\vec{b}$ have higher decision orders than the other variables in making truth assignments), the procedure of Figure 2 imposes no restriction on the variable decision order. This freedom makes it much more effective. On the other hand, since this procedure terminates only when all infeasible solutions are pruned, sometimes its termination may take a long time. Nevertheless its number of SAT solving iterations has the same upper bound as that of Figure 1 for a similar reason.

PROPOSITION 4. *The procedure of Figure 2 for Boolean matching under NP-equivalence terminates within $O(2^{2n})$ iterations.*

Note that the computation of Figure 2 can be understood as performing quantifier elimination of variables $\vec{x}$ and $\vec{y}$ of Formula (5). The resultant formula (in terms of variables $\vec{a}, \vec{b}$) characterizes all matching solutions.

3.2 Boolean Matching with Abstraction

We propose a new preprocessing method and introduce abstract Boolean matching.

DEFINITION 1. *Given a Boolean function $f(\vec{x})$, a variable subset $X^* \subseteq X$, and a variable z, the* **abstract function** *f^* of f with respect to the (X^*, z)-***abstraction** *α is defined to be $f^*(\vec{x^*}, z) = f(\alpha(x_1), \ldots, \alpha(x_n))$ for*

$$\alpha(x_i) = \begin{cases} x_i, & \text{if } x_i \in X^*; \\ z, & \text{otherwise.} \end{cases}$$

Variables X^* are referred to as the **concrete variables***, and z is the* **abstract variable***.*

Given two functions $f(\vec{x})$ and $g(\vec{y})$, let f^* be the abstract function of f with respect to the (X^*, z)-abstraction α. The **abstract Boolean matching** determines if $f^*(\vec{x^*}, z)$ and $g(\vec{y})$ can be equivalent under the variable mapping $\vec{y} = \alpha \circ \nu \circ \pi(\vec{x})$ for some $\nu \circ \pi$. (The function α is phase-preserving, i.e., $\alpha(\neg x_i) = \neg\alpha(x_i)$.)

To represent $\vec{y} = \alpha \circ \nu \circ \pi(\vec{x})$, we define the 0-1 matrix:

$$\begin{array}{c} \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{array} \begin{array}{c} \begin{array}{cccccccc} x_1^* & \neg x_1^* & \cdots & x_k^* & \neg x_k^* & z & \neg z \end{array} \\ \begin{pmatrix} a_{11} & b_{11} & \cdots & a_{1k} & b_{1k} & a_{1(k+1)} & b_{1(k+1)} \\ a_{21} & b_{21} & \cdots & a_{2k} & b_{2k} & a_{2(k+1)} & b_{2(k+1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ a_{n1} & b_{n1} & \cdots & a_{nk} & b_{nk} & a_{n(k+1)} & b_{n(k+1)} \end{pmatrix} \end{array} \quad (7)$$

with

$$\sum_{j=1}^{k+1}(a_{ij} + b_{ij}) = 1 \qquad \text{for } i = 1, \ldots, n, \text{ and} \quad (8)$$

$$\sum_{i=1}^{n}(a_{ij} + b_{ij}) = 1 \qquad \text{for } j = 1, \ldots, k. \quad (9)$$

Furthermore, let $a_{ij} = 1$ and $b_{ij} = 1$ indicate $(y_i \equiv x_j)$ and $(y_i \equiv \neg x_j)$, respectively, for $j = 1, \ldots, k$; let $a_{i(k+1)} = 1$ and $b_{i(k+1)} = 1$ indicate $(y_i \equiv z)$ and $(y_i \equiv \neg z)$, respectively. All of these constraints can be expressed with a Boolean formula and a solution to it corresponds to some legal $\nu \circ \pi$ with respect to the abstraction α.[2]

To solve the abstract Boolean matching, the procedures of Figures 1 and 2 can be applied with a similar learning mechanism. Again let $\Phi^{\langle i \rangle}$ be the formula characterizing the remaining matching solutions after the ith iteration.

PROPOSITION 5. *For abstract Boolean matching of $f^*(\vec{x^*}, z)$ and $g(\vec{y})$ under NP-equivalence, if $f(\alpha(\vec{u})) \neq g(\vec{v})$ for $\vec{u} \in [\![\vec{x}]\!]$ and $\vec{v} \in [\![\vec{y}]\!]$ with $\vec{v} = \alpha \circ \nu \circ \pi(\vec{u})$ for some $\alpha \circ \nu \circ \pi$ satisfying $\Phi^{\langle i \rangle}$, then conjuncting $\Phi^{\langle i \rangle}$ with the clause $\kappa = \bigvee_{i=1}^{n} \bigvee_{j=1}^{k+1} l_{ij}$ for literals*

$$l_{ij} = \begin{cases} a_{ij}, & \text{if } v_i \neq u_j^*, \\ b_{ij}, & \text{otherwise,} \end{cases}$$

where $\vec{u^*} \in [\![(\vec{x^*}, z)]\!]$ for $f^*(\vec{u^*}) = f(\alpha(\vec{u}))$, excludes from $\Phi^{\langle i \rangle}$ exactly the mappings $\{\alpha \circ \nu' \circ \pi' \mid \alpha \circ \nu' \circ \pi'(\vec{u}) = \alpha \circ \nu \circ \pi(\vec{u})\}$.

So a learned clause is of size $n(k+1)$.

The abstract Boolean matching of f^* and g is useful for two reasons. First, the computation is simplified (f^* is simpler than f and the leaned clauses are shorter). Second, its solutions reveal useful information for the Boolean matching of f and g.

[2]Note that z and $\neg z$ are allowed to map to multiple variables in Y since Eq. (9) imposes no cardinality constraint for $j = k+1$. In contrast, one of x_i^* and $\neg x_i^*$ maps to a unique variable in Y.

PROPOSITION 6. *If a mapping between the concrete variables X^* and some $Y' \subseteq Y$ (with $|X^*| = |Y'|$) is shown infeasible in the abstract Boolean matching of f^* and g, then in the original Boolean matching of f and g any mappings between X and Y having the same sub-mapping between X^* and Y' must be infeasible as well.*

By conducting abstract Boolean matching for different sets of concrete variables, it can be applied as preprocessing to filter out infeasible matching solutions for the original Boolean matching problem.

For the preprocessing purpose, the procedure of Figure 2 is more effective for abstract Boolean matching than that of Figure 1. In fact, the termination condition can be relaxed to quit at any iteration since the remaining matching solutions characterized by $\Phi^{\langle i \rangle}$ in the abstract Boolean matching will be a legitimate over-approximation of the matching solutions in the original Boolean matching.

4. MATCHING FOR P-EQUIVALENCE

Since Boolean matching under P-equivalence is a special case of matching under NP-equivalence, the computation framework of NP-equivalence can be customized for P-equivalence.

To represent $\vec{y} = \pi(\vec{x})$, Matrix (2) for NP-equivalence is applicable by removing the columns indexed by $\neg x_1, \ldots, \neg x_n$. Moreover, the cardinality constraints are the same as Eq. (3) and Eq. (4) but excluding the b_{ij} terms. They can be expressed by a CNF formula (new φ_C) with n^2 variables and $2n(C_2^n + 1) = O(n^3)$ clauses. By asserting $\bigwedge_{i,j=1}^{n}(a_{ij} \Rightarrow (y_i \equiv x_j))$ (new φ_A), legal permutations can be characterized using a Boolean formula. Thus Boolean matching under P-equivalence can be formulated as solving the QBF (5) with the updated φ_C and φ_A.

Similar to the NP-equivalence case, the procedures of Figures 1 and 2 are applicable here. As a matter of fact, leaning for P-equivalence can be made more efficient. Let $\Phi^{\langle i \rangle}$ characterize the remaining matching solutions at the ith iteration.

PROPOSITION 7. *Given two functions $f(\vec{x})$ and $g(\vec{y})$ for Boolean matching under P-equivalence, if $f(\vec{u}) \neq g(\vec{v})$ for $\vec{u} \in [\![\vec{x}]\!]$ and $\vec{v} \in [\![\vec{y}]\!]$ with $\vec{v} = \pi(\vec{u})$ for some π satisfying $\Phi^{\langle i \rangle}$, then conjuncting $\Phi^{\langle i \rangle}$ with the clause $\kappa = \bigvee_{i,j=1}^{n} l_{ij}$ for literals*

$$l_{ij} = \begin{cases} a_{ij}, & \text{if } v_i = 0 \text{ and } u_j = 1; \\ \emptyset, & \text{otherwise,} \end{cases}$$

excludes from $\Phi^{\langle i \rangle}$ exactly the mappings $\{\pi' \mid \pi'(\vec{u}) = \pi(\vec{u})\}$.

Note that the above condition "if $v_i = 0$ and $u_j = 1$" can be equivalently replaced with "if $v_i = 1$ and $u_j = 0$." Clearly for $\vec{u} \in [\![\vec{x}]\!]$ with m u_i's equal to 1, then the corresponding learned clause is of $m(n-m)$ literals.

EXAMPLE 3. *The learned clause corresponding to $f(1,0,1) \neq g(0,1,1)$ can be $(a_{11} \vee a_{13})$ or, equivalently, $(a_{22} \vee a_{32})$.*

The pruning power of a learned clause can be characterized as follows.

PROPOSITION 8. *For Boolean matching under P-equivalence, the clause κ learned from a satisfying solution $\vec{u} \in [\![\vec{x}]\!]$ to $f(\vec{u}) \not\equiv g(\pi(\vec{u}))$ for some π prunes $m!(n-m)!$ infeasible permutations, where m is the number of 1's in $\vec{u}$.*

Observe that the larger the difference between the numbers of 1's and 0's in $\vec{u}$ is, the stronger the pruning power of the learned clause is. Hence it may be beneficial to search satisfying solutions in such biased truth assignments.

For every $\vec{u} \in [\![\vec{x}]\!]$ with k 1's, there are C_k^n possible $\vec{v} \in [\![\vec{y}]\!]$ having the same number of 1's. Hence the number of possible learned clauses and thus learning iterations is upper bounded by $\frac{1}{2}((C_0^n)^2 + (C_1^n)^2 + \cdots + (C_n^n)^2) = \frac{2^{2n} \cdot (n-\frac{1}{2})!}{2\sqrt{\pi} n!} \leq \frac{2^{2n}}{2\sqrt{\pi}}$.

PROPOSITION 9. *The Boolean matching procedures of Figures 1 and 2 for P-equivalence both terminate within $O(2^{2n})$ iterations.*

Preprocessing with abstraction can be pursued in Boolean matching under P-equivalence similar to that for NP-equivalence. We omit the exposition to save space.

5. EXPERIMENTAL RESULTS

BooM was programmed in C language within the ABC [6] package using MiniSAT [10] as the underlying solver. All experiments were conducted on a Linux machine with Xeon 2.5GHz CPU and 26GB RAM.

Circuits from the MCNC, ISCAS89 and ITC99 benchmark suites were chosen. Sequential circuits were converted to combinational ones by the ABC command `comb`. To test the full power of BooM, we make the computation harder by matching each primary output of a circuit independently.[3] A function is matched against its synthesized version with its inputs permuted in a reverse/random order (for P-equivalence checking), and in addition negated randomly (for NP-equivalence checking). Functions with support sizes between 10 and 39 were experimented. Specifically there are 717 functions with an average of 114.74 AIG nodes (ranging from 15 to 2160 nodes) and 23.74 variables.

The baseline preprocessing of BooM to reduce the search space consists of detecting functional properties of NE-symmetry and unateness, and simulating Type 1 and Type 2 vectors of [19]. For two symmetry groups that can be uniquely mapped between the two functions to be matched, BooM breaks the symmetry by assigning an arbitrary variable mapping.

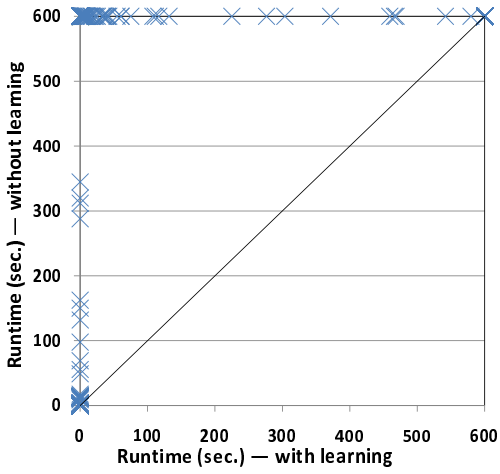

Figure 3: Runtime with and without learning

Figure 3 shows the effect of learning (as an example, in the context of searching all matching solutions under P-equivalence). The x- and y-axes correspond to the runtimes with and without learning, respectively. A spot in the figure corresponds to the result of a function. As most of the spots are above the 45-degree line, learning is evidentally useful.

Figure 4 shows the effect of abstraction (in the same context as Figure 3 with learning applied). The abstraction

[3]When multiple outputs are considered simultaneously, e.g., in [19], many more mutual signatures can be deduced to reduce the search space substantially.

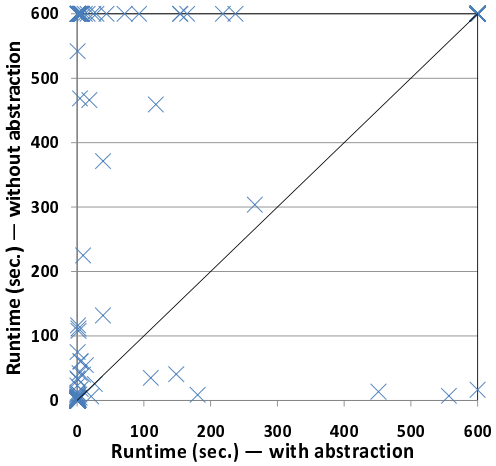

Figure 4: Runtime with and without abstraction

is conducted for enumerating half of all possible abstract matchings with two concrete variables. Only functions with non-empty clauses learned from abstraction are plotted. As can be seen, abstraction achieves clear improvement. Many instances timed out after 600 seconds without abstraction can be effectively resolved with abstraction. Nevertheless, there are a few cases where abstraction does not help.

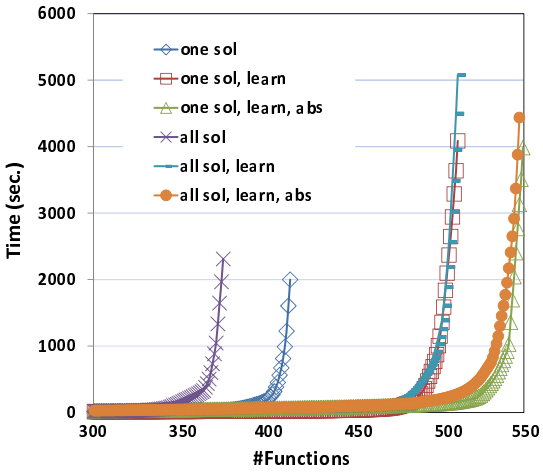

Figure 5: Cumulative runtime for P-equivalence

Figures 5 and 6 show the cumulative runtime for matching under P-equivalence and NP-equivalence, respectively. In these figures, the x- and y-axes are indexed by the cumulative number of solved instances and cumulative runtime, respectively. In the legends, "one sol" and "all sol" indicate the targets of searching one matching solution (Section 3.1.1) and searching all matching solutions (Section 3.1.2), respectively; "learn" indicates learning being applied; "abs" indicates preprocessing using abstraction being turned on. The solved instances under each of these six options were sorted by their runtimes in an ascending order before the accumulation. (Among the solved functions in searching one matching solution using both learning and abstraction, the maximum input sizes are 39 and 38 for matching under P-equivalence and NP-equivalence, respectively.) These two figures reveal, as expected, that matching under NP-equivalence is much harder than that under P-equivalence. Moreover, the performance of searching all matching solutions is comparable to that of searching one solution in the P-equivalence case, but far worse in the NP-equivalence case. One explanation might be that, since the configuration space for NP-equivalence is

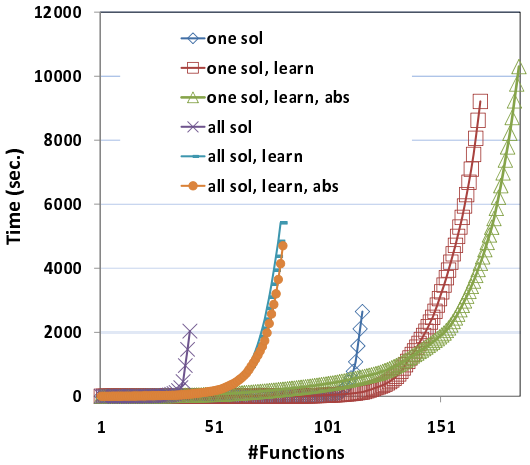

Figure 6: Cumulative runtime for NP-equivalence

much larger, searching all matching solutions requires many more refinement iterations and thus becomes less effective.

In all the above experiments, a matching instance consists of a pair of functions, whose variables are in reverse order. To see the effect of variable ordering, we alternatively prepared matching instances with random order. For searching all matching solutions under P-equivalence, 531 out of the 717 functions can be solved within 600 seconds under both orders, and the corresponding total-runtime ratio of reverse order to random order is 1.00 to 1.06. Since there seems no strong bias when using these two orders, the results under reverse ordering may be more or less representative.

Experience of matching incompletely-specified functions (data not shown due to space limitation) suggested that it takes longer time to solve than matching completely-specified counterparts. The reason can be twofold: First, more assignments on average are tried before the miter constraint is satisfied. Second, the miter constraint becomes more complex in representing the care conditions. Hence the runtime per learning iteration is larger. Nevertheless matching incompletely-specified functions is indeed feasible under the BooM framework.

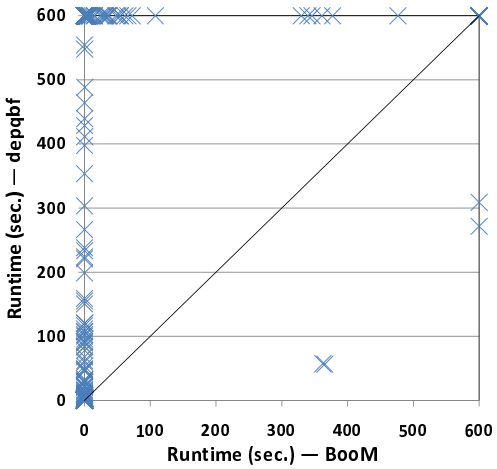

Figure 7: Comparison between BooM and DepQBF

Figure 7 compares BooM and DepQBF [9], a state-of-the-art QBF solver, in searching one matching solution under P-equivalence. The x- and y-axes correspond to the runtimes of BooM and DepQBF, respectively, after the same preprocessing. As can be seen, BooM outperforms DepQBF due to its unique and powerful domain-specific learning.

6. CONCLUSIONS AND FUTURE WORK

We have formulated Boolean matching as QBF solving and exploited domain-specific knowledge for effective search space reduction. A decision procedure BooM, equipped with abstraction and dynamic learning, has been proposed as a generic computation framework for Boolean matching under NPN-equivalence for both completely and incompletely specified functions. Experiments showed promising results. As various Boolean matching techniques can be built and integrated on top of BooM, we anticipate Boolean matching can be made scalable and practical in more applications. Moreover, the success of BooM may suggest that it worths to customize decision procedures for other computation problems.

Acknowledgments

The authors are grateful to Bo-Han Wu for helpful discussion, and National Science Council for grants 96-2221-E-002-278-MY3 and 98-2221-E-030-016.

7. REFERENCES

[1] A. Abdollahi. Signature based Boolean matching in the presence of don't cares. In *Proc. DAC*, pp. 642-647, 2008.

[2] G. Agosta, F. Bruschi, G. Pelosi, and D. Sciuto. A trasnform-parametric approch to Boolean matching. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 6, pp. 805-817, Jun. 2009.

[3] A. Abdollahi and M. Pedram. Symmetry detection and Boolean matching utilizing a signature-based canonical form of Boolean functions. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 6, pp. 1128-1137, Jun. 2008.

[4] M. Agrawal and T. Thierauf. The Boolean isomorphism problem. In *Proc. IEEE Symp. on Foundations of Computer Science*, pp. 422-430, 1996.

[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, Jul. 1997.

[6] Berkeley Logic Synthesis and Verification Group. *ABC: A system for sequential synthesis and verification.* `http://www.eecs.berkeley.edu/~alanmi/abc/`

[7] B. Borchert, D. Ranjan, and F. Stephan. On the computational complexity of some classical equivalence relations on Boolean functions. *Forschungsberichte Mathematische Logik*, Universität Heidelberg, Bericht Nr. 18, Dec. 1995.

[8] J. Cong and Y.-Y. Hwang. Boolean matching for LUT-based logic blocks with applications to architecture evaluation and technology mapping. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 9, pp. 1077-1090, Sep. 2001.

[9] DepQBF: `http://fmv.jku.at/depqbf/`

[10] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT*, pp. 502-518, 2003.

[11] S. Krishnaswamy, H. Ren, N. Modi, and R. Puri. DeltaSyn: An efficient logic-difference optimizer for ECO synthesis. In *Proc. ICCAD*, 2009.

[12] J. Mohnke, P. Molitor, and S. Malik. Limits of using signatures for permutation independent Boolean comparison. In *Proc. ASP-DAC*, pp. 459-464, 1995.

[13] J. Mohnke, P. Molitor, and S. Malik. Application of BDDs in Boolean matching techniques for formal logic combinational verification. *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 1-10, Springer, May 2001.

[14] M. Moskewicz, C. Madigan, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC*, pp. 530-535, 2001.

[15] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. on Computers*, vol. 48, no. 5, pp. 506-521, May 1999.

[16] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, pp. 466-483, 1970.

[17] K.-H. Wang and C.-M. Chan. Incremental learning approach and SAT model for Boolean matching with don't cares. In *Proc. ICCAD*, pp. 234-239, 2007.

[18] Z. Wei, D. Chai, A. Kuehlmann, and A. R. Newton. Fast Boolean matching with don't cares. In *Proc. Int. Symp. on Quality Electronic Design*, pp. 346-351, 2006.

[19] K.-H. Wang, C.-M. Chan, and J.-C. Liu. Simulation and SAT-based Boolean matching for large Boolean networks. In *Proc. DAC*, pp. 396-401, 2009.