Efficient Verification and Synthesis using Design Commonalities

Gitanjali Swamy
Boston Advanced Development Labs
Mentor Graphics, Boston,MA

Stephen Edwards Robert Brayton
University of California at Berkeley
Berkeley, CA

Abstract

In this paper we solve the problem of identifying a "matching" between two logic circuits or "networks". A matching is a functions that maps each gate or "node" in the new circuit into one in the old circuit (if a matching does not exist it maps it to null). We present both an exact and a heuristic way to solve the maximal matching problem. The matching problem does not require any input correspondences. The purpose is to identify structurally identical regions in the networks, and exploit the commonality between them for more efficient verification and synthesis.

Synthesis and verification tools that recognize commonalities both between two versions of the same design, as well within a single design, may be able to outperform their counterparts that do not utilize these commonalities. This work is concerned with detecting structural "matchings" that may be re-utilized.

1 Introduction

We address the problem of finding a high quality matching between two networks. We compare pairs of networks—combinational logic designs represented as directed acyclic graphs whose nodes are generalized (multi-valued, non-deterministic) gates and whose edges are generalized (multi-valued) connecting wires. We look for matchings, functions $M : N \rightarrow N' \cup \{\emptyset\}$ from each node in a new network N to a node in the old network N' or to "unmatched" ($\emptyset$) such that if $M(n) = n'$, then the gates at nodes n and n' are identical (when their inputs are permuted) and their fanins match ($M(n_k) = n'_k$ for corresponding fanins n_k and n'_k). The quality of a matching is the number of matched nodes $q(M) = |\{n \in N | M(n) \neq \emptyset\}|$. We solve the problem of finding the maximum quality matching.

The ability to *reuse parts of the design* during synthesis and verification is our primary motivation for solving this problem. Our application was incremental design analysis, where we have multiple versions of the same design (generated by a compiler that lost the correspondences) and would like to share similar information. However, any application where input correspondences between design are not available is a valid application for this technique. One example is verification under re-timing [8], where latch correspondences get distorted and lost. We may want to avoid re-computation of information shared between the original and retimed design, but we do not have 1-1 correspondences between the inputs and outputs. We can still share functional and structural information between node n and the corresponding matching node $M(n)$. Analysis can be done more efficiently by *identifying unchanged portions of a design and reusing the information* computed for them. Our techniques *may also* be used to *identify common areas within a single design*, allowing common information to be computed efficiently. This is particularly relevant in the context of circuits generated after high-level synthesis, where we may have multiple instantiations of the same module that lead to similar structures in the circuit, and we would like to avoid re-computation for each instantiation.

This common information can be used to create more efficient synthesis and verification algorithms. It must be stressed that even though the experiment we ran used verification to illustrate this technique, the *focus of this work is how to identify the design commonalities.*

In the context of synthesis and verification, a design commonality or *matching* corresponds to structurally identical transitive fanin cones of the design that start at a node and contain all the nodes and wires in its transitive fanin. We choose to identify these because the global function at a node is a function only of its transitive fanins. An example is the transition function [12], used frequently in formal verification and usually computed using BDDs [4]. Identifying matching nodes allows us to compute the new BDD by substituting variables, which can be done efficiently. For example, the BDD for $f(b)$ can be obtained from a BDD for $f(a)$ by substitution, even though a and b are different primary inputs altogether.

The approach we propose *does not require any additional matching information (e.g., correspondences between the primary inputs).* We expect most designs we compare will be the *output of a compiler* that does not usually supply any correspondence information. An alternative would be to use names to guess correspondences, but this is insufficient when names are automatically generated—they are often very sensitive to small changes in a design. Finally, by not assuming input correspondences, our algorithms can be applied to more general problems such as *identifying identical structures* within the same design. Keep in mind that if the input correspondences were available, there are more efficient techniques to solve the problem.

We propose a greedy three-phase algorithm to find a good matching. First, nodes with identical functions are identified. Next, this information is combined with connectivity information to find nodes that have identical structures in their transitive fanins. Finally, the matchings implied by these nodes are combined into a high-quality matching. We use both a greedy heuristic, as well as an exact formulation.

It is not correct to compare Brand et al's [1] work on incremental synthesis with this work, because they require knowledge of input correspondences and can only detect regions that start at the inputs and have the exact same function.

Another relevant piece of work by Burch et al [5] solves a functional matching problem that does not require input correspondence information. However, they are only comparing Boolean functions, and their approach does not generalize to circuit designs. Note that one *sub-problem* in our network matching is node function matching, which could use Burch's approach. However, our main objective is to get a quick matching rather than the exact node function matching. We adopt a similar notion of a semi-canonical form, but our form is simpler (and hence faster) at the expense of some precision. Also, we deal with more general multi-valued functions [2], rather than just binary.

2 Network Matching

We assume the reader is familiar with the following concepts: A **network** N or netlist of logic gates is characterized by a set of **nodes** n or logic gates with three associated functions: $\text{func}(n)$ is the **function** of the node, $\text{fanins}(n) \in \{0, 1, \ldots\}$ is the **number of fanins** of the node, and $\text{fanin}(n, k) \in N, k = \{1, \ldots, \text{fanins}(n)\}$ is the **kth fanin of the node**. The entire set of fanins of fanins etc are denoted as the **transitive fanins** $tf(n)$.

In general, this problem is hard; *it is easy to see that an instance of sub-graph isomorphism can be reduced to an instance of this problem, making it NP-hard.*

Our aim is to find a node in the old network for each node in the new network, with information we can use for its analysis. This information, by assumption, is only a function of the node and its transitive fanin. Thus, the matching node in the old network must have an identical transitive fanin (only up to the inputs). In any case, we can use information computed for a node to get the same information for its matching node, irrespective of the primary inputs involved. For example, the BDD for $f(b)$ can be obtained from a BDD for $f(a)$ by substitution, even though a and b are different primary inputs altogether.

We cannot use the technique of using the simulation signatures of nodes to distinguish them, because *we do not have an input correspondence.* We identify the set of all potentially matching nodes (called candidate pairs) and combine a compatible subset of these to form the matching. In Section 4, we show that the problem of finding the best subset can be reduced to finding a maximal prime compatible. In Section 5, we present a greedy algorithm for finding a good subset.

The following definition characterizes which nodes we might consider matching. Informally, two nodes could match if their functions are identical and their respective fanins could match.

Definition 1 *A pair of nodes*

n_1, n_2 *is a* **candidate pair** *(denoted $n_1 \sim n_2$) if* $\mathrm{func}(n_1) = \mathrm{func}(n_2)$, $\mathrm{fanins}(n_1) = \mathrm{fanins}(n_2)$, *and* $\forall_{k=1,\ldots,\mathrm{fanins}(n_1)} \mathrm{fanin}(n_1, k) \sim \mathrm{fanin}(n_2, k)$. *Note that the correspondence between the fanins is determined by reducing the node function representation to some semi-canonical form, and noting that in that form, the ith variable for (canonical) node function for n* must *correspond with the ith variable for the (canonical) node function for n'.*

This is of course an approximation, since there may be several permutations of fanins where $func(n_1) = func(n_2)$. Note that this definition implies that all primary inputs may match with each other. We add the caveat that the primary inputs may match provided they can take the same set of values, i.e. a primary input that can take values $0, 1, 2$ cannot match with a primary input that takes values $0, 1, 2, 3, 4, 5$.

Not all candidate pairs lead to consistent matchings. Specifically, it may be necessary to match a node in the new network to two or more nodes in the old network simultaneously. This is particularly nonsensical in the case of zero-fanin nodes, which represent inputs to the network. Figure 1 depicts a contradictory situation.

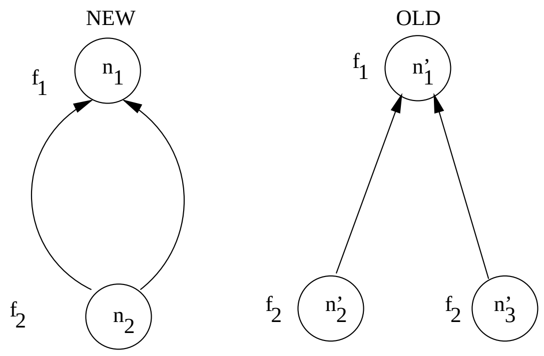

Figure 1. A candidate pair ($n_1 \sim n_1'$) with no consistent matching.

Formally, the consistency constraint requires a matching to be a function mapping each node in the new network either to a matched node in the old network, or to "unmatched," represented as $\emptyset$.

Definition 2 *Given two networks N (the new network) and N' (the old network), a* **matching** *is a function $M : N \to N' \cup \{\emptyset\}$ such that $M(n) \neq \emptyset$ implies ($n \sim M(n)$ and $\forall k = 1, \ldots, \mathrm{fanins}(n)$. $M(\mathrm{fanin}(n, k)) = \mathrm{fanin}(M(n), k)$).*

Note: This definition implies that if $M(n) \not\models \phi$, then $\forall n_a \in \mathrm{tf}(n), M(n_a) \not\models \phi$.

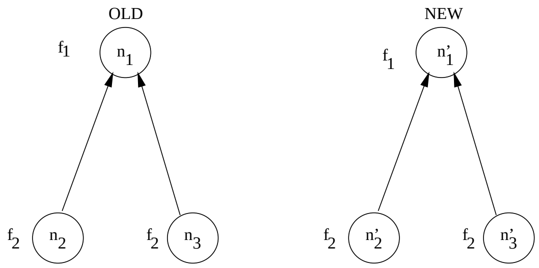

Figure 2. A matching with $\mathrm{q}(M) = 3$

Our objective is to find a matching that maximizes the number of matched nodes (called the quality of the match), i.e. those for which $M(n) \not\models \phi$.

Definition 3 *The* **quality** *of a matching M is the number of matched nodes, i.e.,* $\mathrm{q}(M) = |\{n \mid M(n) \neq \emptyset\}|$.

Definition 4 *If it exists, the* **implied matching** *of a candidate pair $n_1 \sim n_2$ is*

$$\begin{aligned} M(n_1) &= n_2 \\ \forall_k M(\mathrm{fanin}(n_a, k)) &= \mathrm{fanin}(M(n_a), k),\ n_a \in \mathrm{tf}(n_1) \cup \{n_1\} \\ M(n) &= \emptyset,\ n \notin \mathrm{tf}(n_1) \end{aligned}$$

Theorem 2.1 *An implied matching is a matching.*

Proof.

1. $\forall k = 1, \ldots, \mathrm{fanins}(n), M(\mathrm{fanin}(n, k)) = \mathrm{fanin}(M(n), k)$.

2. M is a function.

□

We will be combining implied matchings to form bigger matchings, but some pairs of implied matchings—those that map a node in the new network to two different nodes in the old—cannot be combined. We need a formal definition of which matchings can be merged:

Definition 5 *A pair of matchings M_1 and M_2 are* **compatible** *(written $M_1 \rightleftharpoons M_2$) if $(M_1(n) \neq \emptyset) \wedge (M_2(n) \neq \emptyset) \Rightarrow M_1(n) = M_2(n)$.*

Note that compatibility is not transitive; i.e. $M_1 \rightleftharpoons M_2$, and $M_2 \rightleftharpoons M_3$, does not imply that $M_1 \rightleftharpoons M_3$.

Definition 6 *The* **merge** *of two matchings M_1 and M_2, written $M_1 + M_2$, is the function*

$$(M_1 + M_2)(n) = \begin{cases} M_2(n) & \text{if } M_1(n) = \emptyset \\ M_1(n) & \text{otherwise} \end{cases}$$

Lemma 2.2 *If $M_1 \rightleftharpoons M_2$, then $M_1 + M_2$ is a matching and $M_1 + M_2 = M_2 + M_1$, i.e. merging is commutative. Moreover, if in addition $M_2 \rightleftharpoons M_3$ and $M_1 \rightleftharpoons M_3$, then $(M_1+M_2)+M_3 = M_1+(M_2+M_3)$, i.e. merging is associative.*

Proof. $M_1 \rightleftharpoons M_2 \Leftrightarrow \forall_n (M_1(n) \neq \emptyset) \cdot (M_2(n) \neq \emptyset) \Rightarrow M_1(n) = M_2(n)$.

$$(M_1 + M_2)(n) = \begin{cases} M_2(n) & \text{if } M_1(n) = \emptyset \\ M_1(n) & \text{if } M_1(n) \neq \emptyset \end{cases}$$

$$(M_2 + M_1)(n) = \begin{cases} M_1(n) & \text{if } M_2(n) = \emptyset \\ M_2(n) & \text{if } M_2(n) \neq \emptyset \end{cases}$$

1. if $M_1 \neq \emptyset$, $M_2 \neq \emptyset$.
   $\Rightarrow M_1 + M_2 = M_1 = M_2 = M_2 + M_1$.

2. if $M_1 = \emptyset$, $M_2 \neq \emptyset$.
   $\Rightarrow M_1 + M_2 = M_2 = M_2 + M_1$.

3. if $M_1 \neq \emptyset$, $M_2 = \emptyset$.
   $\Rightarrow M_1 + M_2 = M_1 = M_2 + M_1$.

4. if $M_1 = \emptyset$, $M_2 = \emptyset$.
   $\Rightarrow M_1 + M_2 = \emptyset = M_2 + M_1$.

$\Rightarrow M_1 + M_2 = M_2 + M_1$. Associativity proved in a similar manner, i.e. by enumerating all possibilities. □

Lemma 2.3 *Merging only improves quality, i.e., if $M_1 \rightleftharpoons M_2$, then* q$(M_1), q(M_2) \leq q(M_1 + M_2)$.

Proof. Assume not.
$\Rightarrow \exists n \; st \; (M_1(n) \neq \emptyset) \cdot (M_1 + M_2)(n) = \emptyset$.
$M_1(n) \neq \emptyset \Rightarrow (M_1 + M_2)(n) = M_1(n) \neq \emptyset$.
$\Rightarrow (M_1 + M_2)(n) \neq \emptyset$.
A contradiction, hence $\nexists n \; st \; (M_1(n) \neq \emptyset) \cdot ((M_1 + M_2)(n) = \emptyset$.
$\Rightarrow q(M_1) \leq q(M_1 + M_2)$. □

Partition nodes in both networks by function
Refine this partition s.t. all nodes in a bucket have fanins in the same buckets
Form all candidate pairs by considering all pairs of nodes in each bucket
Sort the candidate pairs by the number of nodes in their transitive fanin

Figure 3. Identifying compatible nodes.

3 Determining Matchings: A Refinement Algorithm

In order to determine the entire set of implied matchings, we use the following iterative algorithm. We begin by assuming all nodes whose node functions are matched to be matched. We implement this algorithm with a hash table. Nodes with the same node function are put into the same initial "bucket" in the hash table. The canonical form of the node function imposes a certain order on the fanins of the node. If two node functions in canonical form are equal, then the fanins node corresponding to ith variable of the node function, must correspond. We refine the node matchings iteratively, by "un-matching" two nodes, if some of their corresponding fanins are un-matched. We accomplish this by re-bucketing each node in the hash table. At each iteration, the new bucket signature of a node consists of its table signature (canonical form) and the bucket numbers of its fanins (in the order imposed by their node function tables). Thus, if at some iteration, any nodes in the same bucket have corresponding fanins in different buckets, then after that iteration, these nodes get put into different buckets.

This algorithm is similar to the algorithm for the computation of equivalent states in an FSM [6], [12]. After this refinement, all pairs of nodes in a bucket are candidates. The algorithm is shown in Figure 3.

Note that at th ith iteration of this algorithm, nodes that match up to at least i levels of fanin are identified. Thus, though we have described a procedure that matches entire cones, this procedure can be modified to match sub-regions by restricting the number of iterations of the refinement procedure, or keeping track of all buckets seen during the refinement process.

4 An Exact Formulation

Once we have a set of consistent matchings (Section 3), we address the problem of finding a maximum compatible matching exactly.

Lemma 2.3 indicates that merging compatible matchings gives higher quality matchings. In this section, we use this idea to exactly characterize the problem of finding the maximal quality matching. We show that the maximal matching is a "prime" matching—one for which merging in other matchings is either impossible or unproductive.

Lemma 4.1 *If M is the sum of a finite number of compatible implied matchings then it is a matching, i.e.,* $\forall_{i,j} M_i \rightleftharpoons M_j$ *and* $M = M_1 + M_2 + \cdots + M_k \Rightarrow$ *M is a matching .*

Proof. Follows from the definition of matching, implied matching, and Lemmas 2.2. □

We can define a dominance relation [7] [11] as follows:

Definition 7 *A matching M_1* **dominates** *a matching M_2 (written $M_1 \geq M_2$) if $M_1 \rightleftharpoons M_2$ and $M_1 + M_2 = M_1$.*

Definition 8 *A* **prime** *matching is one that is not dominated by any other matching.*

Lemma 4.2 *If M_1 is a prime matching, and $M_1 \geq M_2$, then* $\mathrm{q}(M_1) \geq \mathrm{q}(M_2)$).

Proof. Since $M_1 \geq M_2$, $M_1 \rightleftharpoons M_2 \Rightarrow M_1 = M_1 + M_2$.
Lemma 2.3 implies $\mathrm{q}(M_2) \leq \mathrm{q}(M_1 + M_2)$. Since $M_1 + M_2 = M_1$, it follows that $\mathrm{q}(M_2) \leq \mathrm{q}(M_1)$. □

We can reduce maximal or prime matching to a prime generation problem in the following manner.

1. Associate a Boolean variable u_i with each matching M_i. $u_i = 1$ implies M_i is part of the given matching.

2. For each pair of matchings M_i and M_j that are not compatible $M_i \not\rightleftharpoons M_j$, construct a clause $(\overline{u_i} + \overline{u_j})$. This means either M_i must not be in the partition or M_j must not be in the partition.

3. logically AND all such clauses to get a function $f(u)$.

4. A prime of function $f(u)$ corresponds to a compatible set of matchings. The maximal prime corresponds to a maximal matching.

Theorem 4.3 *A maximum matching is a prime matching and can be built from a set of compatible implied matchings.*

Proof. Follows from Lemmas 4.1 and 4.2. □

Thus, from the above the problem of finding the maximum matching is one of finding the maximum quality prime. We can do this naively by enumerating each prime matching and calculating its quality (in actuality, we implement a slightly more efficient procedure). However, since the number of primes of a set of n elements is $O(3^n/n)$ [10] and n can be $O(N^2)$, where N is the number of nodes in each network, it is often impractical to explicitly search the entire set of primes. This worst case comes when the network consists of a set of zero-fanin nodes with identical functions.

5 A Greedy Algorithm

The exact method cannot handle large examples; we extend the scope of the examples by using the following heuristic algorithm. Our heuristic algorithm finds the set of all candidate pairs with implied matchings and merges them greedily, trying the highest quality ones first.

First we used the refinement procedure of Section 3 to identify candidate pairs. Once the candidate pairs are identified, we build a matching by merging together compatible implied matchings. We consider candidate pairs one at a time, starting with those with the largest number of nodes in their transitive fanins, and "grow" a matching by merging each compatible implied matching.

The entire algorithm is shown in Figure 4. In Section 7, we report the performance of our implementation of this algorithm against the exact algorithm.

6 Table Matching: Matching Node Functions

In this section, we discuss how to identify whether two node functions are identical if we do not have an input correspondence. This is known as *Boolean matching*, and is a well studied problem.

Partition nodes in both networks by function
Refine this partition s.t. all nodes in a bucket have fanins in the same buckets
Form all candidate pairs by considering all pairs of nodes in each bucket
Sort the candidate pairs by the number of nodes in their transitive fanin
$M(n) = \emptyset$, the empty matching
for M_i largest to M_i smallest
  if $M \rightleftharpoons M_i$
    $M = M + M_i$
RETURNM

Figure 4. The greedy matching algorithm.

For our experiment, we are looking for a quick estimator of whether two node functions, represented as *node function tables* match.

The nodes in our networks have discrete-valued functions (a generalization of Boolean functions) associated with them. These are represented in BLIF-MV-style tables [2], such as that in Figure 5. Each column on the left represents an input variable, and each row is a pattern that, when the inputs match it, produces the output in the rightmost column. Each entry is either a single value (e.g., 3), a set of values (e.g., $1, 2, 5$), or the set of all values (i.e., "–"). Note that BLIF-MV permits symbolic values of the form $red, blue, greeen$, which are represented as the values $0, 1, 2$.

Figure 5 represents a function $f(x_1, x_2, x_3)$ that is 3 when $x_1 = 0$ and $x_2 = 2$ or 3, or when $x_2 = 1$; is 0 when $x_1 = 1$, $x_2 = 0$, and $x_3 = 1$; and is 1 default.

x_1	x_2	x_3	f
0	2,3	–	3
–	1	–	3
1	0	1	0
default			1

Figure 5. A multi-valued table. x_1**,** x_2**, and** x_3 **are the input variables.**

We want to be able to quickly identify tables that compute the same function. Transforming each table into a *permutation-invariant canonical form* is an approximate approach to solving this problem; different tables that are not equivalent modulo permutations may also compute the same function. Computing a canonical form (modulo all permutations) is much more expensive([5]); in the interests of quick computation, we have opted for this simpler semi-canonical form. For example, the tables shown in Figures 5 and 6 are essentially identical modulo a row column permutation, and there is an identical permutation semi-canonical form for both of them, which can be used to identify this.

y_1	y_2	y_3	f
–	–	1	3
–	0	2,3	3
1	1	0	0
default			1

Figure 6. A multi-valued table. y_1**,** y_2**, and** y_3 **are the input variables.**

Definition 9 *Two tables are* **permutation equivalent** *if one can transformed to the other by permuting the rows and columns.*

We assume that the values in each entry are always ordered, so that we do not have to distinguish between $2, 3$ and $3, 2$. To make this entry compact, we use ordered lists of ranges, i.e. $2 - 5, 7 - 8$, to represent each entry.

Definition 10 *A function is* **canonicalizing** *iff it maps all permutation-equivalent tables to a single table, which is called the* **permutation-invariant canonical form** *of the table.*

A function is canonicalizing if it imposes a permutation-invariant total order on rows and columns and then sorts the rows and columns based on this. Finding such a total order is difficult and expensive, so we resort to an order that is partial for certain tables. We count the number of times a particular value appears in the entries in a row or column and order the rows and columns based on this sum. The reason we use this "addition" of the number of times a value occurs in a column as a hash function is because we need a permutation invariant canonical form.

Consider the table in Figure 7. If we order the rows and columns according the number of 1's that

$$
\begin{array}{cc}
 & \sum = \\
\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix} & \begin{array}{c} 2 \\ 3 \\ 1 \end{array} \\
\sum = \quad \begin{array}{ccc} 3 & 2 & 1 \end{array} &
\end{array}
$$

Figure 7. A simple table annotated with the number of 1's in each row and column.

appear in each row and column, we obtain the table in Figure 8. We were fortunate in this example, since the number of 1's in each row and column is different, but in general, this strategy only produces semi-canonical tables.

$$
\begin{array}{cc}
 & \sum = \\
\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} & \begin{array}{c} 1 \\ 2 \\ 3 \end{array} \\
\sum = \quad \begin{array}{ccc} 1 & 2 & 3 \end{array} &
\end{array}
$$

Figure 8. The table in canonical form

We can extend these ideas to tables with set-valued entries by converting each entry to an integer. First, each set is transformed to a vector of 0's and 1's. Each 1 represents the presence of a value in the set; each 0 represents the absence, e.g., the entry $2, 3$ would be represented as a vector (1100). A bitwise sum of all such vectors in a row or column (zero-extending them if necessary) gives a vector than can be used to impose a partial order. E.g. The bitwise sum of $(2, 3) = (1100)$ and $(0, 1, 2) = (0111)$ is (1211). (1211) denotes that in the given column there is one 0 value, one 1 value, two 2 values and one 3 value.

These vectors can be transformed to integers to make them easier to manipulate.

Intuition

Note that in a table with n rows and m columns, the total number of 1's in a position in a column cannot exceed n. Similarly, the total number of 1's in a row cannot exceed m. By transforming these vectors to base $b = \max\{m, n\} + 1$ integers, we can sum the integers in a row or column, and still ensure that each column sum only includes information about that column (*no carry between (value) positions*). For example, if each entry in a column is the entry $2 = (0100)$, and there are 15 columns. The bitwise sum for the column is $0F00$; F denotes 15 in base 16. If we were to represent the number in base 10, then the sum would be (1500), and due the carry we cannot distinguish between fifteen 2 entries versus one 3 and five 2 entries. Under this representation permutation equivalent rows or columns have the same sum. This may result in some ambiguity. Consider the two tables shown in Figure 9; both rows of the given tables have the same sum, and hence are indistinguishable. If this ambiguity is never resolved, then these two rows will never be interchanged. Thus, the fact that the two tables are identical will not be detected. This issue can be resolved by using a secondary tie breaker like the position of the first 1 entry. In general, this problem is part of a larger problem of "symmetries" [9].

$$
\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}
$$

$$
\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}
$$

Figure 9. Identical tables

Definition 11 *For a table with n rows and m columns, let m_j be the maximum value of the input variable in column j, and let $E_{ij}(k)$ be 1 if the entry in row i and column j contains the value k and 0 otherwise. The numerical representation of this table is an $n \times m$ matrix T with entries*

$$
t_{ij} = \sum_{k=0}^{m_j} b^k E_{ij}(k)
$$

It is clear that each subset of values at a table entry has unique encoding t_{ij}. Figure 10 shows the table of Figure 5 converted to a matrix of natural numbers. For this table, $(1 + \max\{m, n\}) = 4$. As an example, the entry $2, 3$ is converted to a base four number: $t_{1,2} = 4^0 \cdot 0 + 4^1 \cdot 0 + 4^2 \cdot 1 + 4^3 \cdot 1 = 80$.

Definition 12 *In an $m \times n$ table (t_{ij}), a row i is* **before** *row k if $\sum_{j=1}^{n} t_{ij} < \sum_{j=1}^{n} t_{kj}$. A column j is* **before** *a column k if $\sum_{i=1}^{m} t_{ij} < \sum_{i=1}^{m} t_{ik}$.*

$$\begin{array}{cc} & \sum = \\ & \begin{pmatrix} 1 & 80 & 5 \\ 5 & 4 & 5 \\ 4 & 1 & 4 \end{pmatrix} \begin{array}{c} 86 \\ 14 \\ 9 \end{array} \\ \sum = & \begin{array}{ccc} 10 & 85 & 14 \end{array} \end{array}$$

Figure 10. The table converted to a matrix of natural numbers.

$$\begin{pmatrix} 4 & 4 & 1 \\ 5 & 5 & 4 \\ 1 & 5 & 80 \end{pmatrix}$$

Figure 11. The table in semi-canonical form

Definition 13 *The* **semi-canonical form** *of a table t_{ij} is a permutation of the rows and columns of t_{ij} such that if row i is before row k then $i < k$, and if column j is before column k then $j < k$.*

Figure 11 shows the table in Figure 10 converted to semi-canonical form.

Theorem 6.1 *A table in semi-canonical form represents the same function as the original table under some permutation of variables.*

Hence two tables with the same semi-canonical form represent the same discrete function.

7 Experiments and Results

We have implemented the algorithms described in the VIS [3] environment.

In order to to test our procedure, we designed the following experiment. We assume that the design has been read in, and the designer has computed the output function BDDs of each node (as functions of the primary inputs). At this point the designer modifies the original design by either changing the functionality, or just re-optimizing the hardware for some other objective. The designer would like to use the BDDs computed for the old network to efficiently compute the BDDs in the new network. Obviously, we assume that there is a sufficient amount of structural similarity between the old and the network design. To emulate a design change, we took MCNC, ISCAS and VIS benchmark examples and modified them to obtain a circuit called "new". The original benchmark spec corresponds to the "old" design.

Example	Non-Inc Time	Inc Time	Match Time	Total Time
bigkey	1	0.183	1.65	1.883
cordic_latches	2.367	0.066	1.7	1.766
clma	11.6	0.8	11.78	12.68
clmb	11.45	0.8	10.45	11.25
des	2.884	0.017	1.967	1.984
i10	13.334	0.067	1.867	1.934
minmax10	800.734	0.2	0.35	0.55
minmax12	352.634	0.25	0.467	0.717
mm9a	27.034	0.033	0.35	0.383
mm9b	526.0	0.2	0.367	0.567
pair	1.434	0.884	0.466	1.35
s13207	1.6	0.217	18.734	18.941
s1423	1.783	0.133	0.317	0.315
s15850	31.617	0.267	12.317	12.584
s38584	10.85	1.35	138.434	139.784

Table 2. Incremental Vs. Non_Incremental Update

As an experiment we built the function BDDs associated with the "old" design. This is done recursively, by building the BDD at each node as a function of the BDDs of its fanin nodes. Next, we ran the matching algorithm on the old and new designs. If there existed a match from a node in the new network, to the old, we re-used the BDD for the old node by merely substituting the old network BDD variables with the corresponding BDD variables in the new network. If there was no match, we re-computed the BDD by using the BDDs computed for the fanin nodes of the new node. We reported time for this incremental computation (Inc Time) as well as the time for computing the matching (Match Time). We also built the BDDs for the new network from scratch, and reported this non-incremental time (Non-Inc Time).

Table 1 reports the quality of the matching Vs. the time to match the examples. Columns 2 and 3 list the number of inputs and outputs in the circuit respectively. The outputs include both the primary outputs and latch inputs for non-combinational cir-

Example	♯ Inputs	♯ Outputs	♯ Nodes Total	♯ Nodes in Match	Initial Time	Refine Time	Match Time
bigkey	262	197	1369	791	0.317	0.033	1.567
clma	382	115	11382	10973	4.766	3.534	11.783
clmb	382	33	10842	10407	4.634	3.416	10.45
cm163a	16	38	68	11	0.017	0	0.017
cordic_latches	23	2	3468	2873	0.35	0.267	1.617
i10	257	224	2754	2750	0.284	0.6	1.734
minmax10	13	40	723	87	0.033	0.117	0.15
minmax12	15	48	914	104	0.066	0.15	0.233
mm9a	12	36	830	637	0.05	0.05	0.316
mm9b	12	35	714	106	0.067	0.083	0.167
s13207	31	790	10065	8713	0.75	1.333	18.583
s1423	17	79	1199	298	0.1	0.083	0.317
s1488	8	25	711	97	0.084	0.083	0.184
s1494	8	25	658	34	0.083	0.083	0.183
s15850	14	683	11591	10272	0.933	1.684	12.183
s38584	12	1730	23775	20839	5.767	7.267	138.434

Table 1. Quality and Time to Match

cuits. Columns 4 and 5 list the total and matched number of nodes in the network respectively. The matching times are listed by its component; i.e. time to get the initial matching(Initial Time), time to refine the partition(Refine Time), and time to generate matching in Column 6, 7 respectively, as well as the total time to match (Match Time = initial + refine +time to generate and evaluate the quality of the entire matching cones), in Column 8. Since we used an explicit matching algorithm, it is rightly observed that as the size of the matching increases so does the time to match. The dominant portion of the time appears to be spent in generating the matching rather than the refinement or initial time.

Table 2 reports the times for the non-incremental BDD computation (Column 2) Vs. the incremental BDD computation (Column 3) and total matching time (Column 4). The times for incremental BDD computation alone were always better than the non-incremental time (obviously using previously computed information is better than no information). However, when we add in the matching time, this is not always the case.

Of the reported example (we only considered those with more than 1 sec of CPU time for non-incremental BDD building), most have significantly better total times for the incremental procedure (match time + incremental time) as compared to the non-incremental procedure. Only 2 had significantly worse time for the incremental method, 3 had approximately equal times and the rest always reported better times (incremental + matching) for the incremental method.

We also report the results on the exact computation (Section 4)as compared to the heuristic (Section 5). The exact method ran out of memory much faster, and hence we were only able to deal with small examples with the exact method. However, Table 3 shows that for examples where the exact method could complete, the heuristic answers were almost always the same.

We only report examples with significant time to build BDDs with the given order. Though our techniques extend to multi-valued examples. We were not able to find multi-valued examples in our set, with large enough BDD time, so their results are not significant and have not been reported.

We have shown that for small examples the exact answer is almost identical to our heuristic. This demonstrates the effectiveness of our heuristic.

We examined the one example where the matching time far exceeded the non-incremental time, and found that the cause of this problem was the large

Example	Heuristic ♯ Nodes in Matching	Exact ♯ Nodes in Matching
apex7	12	12
bbsse	23	23
c8	15	16
cm163a	11	11
i2	48	48
mark1	18	18
minmax10	87	87
minmax12	104	104
mult32b	253	253
term1	62	62

Table 3. Exact Vs. Heuristic Common Substructures

symmetry in the circuit coupled with the large size of the circuit. There were many possible matchings, and examining them all, while determining the qualities of matchings was expensive. As part of future work, the work of Malik [9] to detect symmetries could be used to speed up our computation. We found that as we increased the size of the example, the matching time increased significantly. This is due to our explicit formulation of the matching algorithm. As future work an implicit formulation of the matching algorithms can used to overcome some of the size limitations (implicit prime generation).

Our techniques could be extended to deal with matching arbitrary sections of the network, rather than the entire transitive fanin cone. One application would be finding structurally identical sections within a single network, so that information computed at one section may be re-used for another structurally identical portion.

References

[1] D. Brand, A. Drumm, S. Kundu, and P. Narain. Incremental Synthesis. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 14–18, Nov. 1994.

[2] R. K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R. P. Kurshan, S. Malik, A. L. Sangiovanni-Vincentelli, E. M. Sentovich, T. Shiple, K. J. Singh, and H.-Y. Wang. BLIF-MV: An Interchange Format for Design Verification and Synthesis. Technical Report UCB/ERL M91/97, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Nov. 1991.

[3] R. K. Brayton et al. VIS: A System for Verification and Synthesis. In *Proc. of the Conf. on Computer-Aided Verification*, pages 428–432, 1996.

[4] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, C-35:677–691, Aug. 1986.

[5] J. Burch and D. Long. Efficient boolean function matching. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 408–411, November 1992.

[6] J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations. Proceedings of an International Symposium on the Theory of Machines and Computations.*, pages 189–196, Haifa, Isreal, 1971. Academic Press.

[7] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A Fully Implicit Algorithm for Exact State Minimization. In *Proc. of the Design Automation Conf.*, pages 684–690, June 1994.

[8] C. E. Leiserson and J. B. Saxe. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, Spring 1983.

[9] S. Malik, J. Mohnke, and P. Molitor. Limits of Using Signatures for Permutation Indepedant Boolean Matching. In *Proc. Intl. Workshop on Logic Synthesis*, Tahoe, May 1995.

[10] E. J. McClusky. Minimization of Boolean Functions. *Bell System Technical Journal*, 35, 1956.

[11] G. M. Swamy, P. Mcgeer, and R. K. Brayton. An Exact Logic minimizer using BDD based Methods . Technical Report "Masters Thesis" UCB/ERL M93/94, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1993.

[12] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 130–133, Nov. 1990.