# Identifying Common Substructure for Incremental Methods

Stephen A. Edwards  Gitanjali M. Swamy  Robert K. Brayton
{sedwards,gms,brayton}@eecs.berkeley.edu
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

May 6, 1996

**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.

We apply this solution to the problem of incremental design. Logic design is usually an iterative process where errors are corrected and optimizations performed repeatedly. A designer rectifies, re-optimizes, and rechecks a design many times. In practice, it is common for small, incremental changes to be made to the design, rather than changing the entirety of the design. Currently, each time the system is modified, the entire set of computations (synthesis,verification) are repeated from the beginning. This results in unneces-

sary re-computation of information, which can be avoided by re-using results of a previous iteration and information about changes to the system.

Synthesis and verification tools that recognize these sequences of slightly-differing inputs may be able to outperform their counterparts that discard all previous work. This work is concerned with detecting what information has changed in a design, and what information 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, a function $M : N \rightarrow N' \cup \{\phi\}$ from each node in a new network $N$ to a node in an old network $N'$ or to "unmatched" ($\phi$) 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 $q(M) = |\{n \in N | M(n) \neq \phi\}|$. We solve the problem of finding the maximum quality matching.

The ability to reuse information is the primary motivation for solving this problem. One application, incremental design analysis, stems from the iterative nature of design. A designer usually wants to analyze each version of a design (with, e.g, a formal verification check). Analysis can be done more efficiently by identifying unchanged portions of a design and reusing the information computed for them. If we have a matching $M$, we can reuse information for each node $n$ where $M(n) \neq \phi$. Our techniques may also be used to identify common areas within a single design, allowing common information to be computed efficiently. Another application is incremental synthesis, where the aim is to preserve as much of the old design as possible.

A 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 fanin. An example is the transition function [1], used frequently in formal verification and usually computed using BDDs [2]. Identifying matching nodes

allows us to compute the new BDD by substituting variables, which can be done efficiently. Our approach 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 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 portions within the same design.

We propose a greedy three-phase algorithm to find a good matching. Initially, nodes with identical functions are identified. This information is then combined with connectivity information to find nodes that have identical structures in their transitive fanins. Finally, the matchings implied by these nodes are greedily combined into a high-quality matching.

This paper is organized as follows. Section 2 describes previous and related work on the problem. Section 3 describes our approach to the gate function (node) matching problem and Section 4 contains our exact and heuristic solutions to the network (structural) matching problem. We present both an exact formulation (Section 4.3) and a greedy algorithm that works well in practice (Section 4.4). Section 5 describes the results of some experiments on the algorithms and presents our conclusions.

## 2 Previous Work

Other approaches to incremental synthesis rely on knowing input correspondences. Brand et al.'s [3, 4] work on incremental synthesis identifies regions of commonality similar to our own, but they require knowledge of input correspondences and

can only detect regions that start at the inputs.

Burch et al. [5] solve 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. 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 [6], rather than just binary.

The techniques presented here can be used to drive the incremental verification algorithms of Swamy et a.l [7] [8] and Sokolosky et al. [9]. These use information about the similarities between two designs to speed up the verification process.

# 3 Table Matching

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 [6], such as that in Figure 1. 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 $0,1,2$. We ignore the fact that $red, blue, greeen$ can also be represented as $2,1,0$.

Figure 1 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.

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

| $x_1$ | $x_2$ | $x_3$ | $f$ |
|---|---|---|---|
| 0 | 2,3 | – | 3 |
| – | 1 | – | 3 |
| 1 | 0 | 1 | 0 |
| | default | | 1 |

Figure 1: A multi-valued table. $x_1$, $x_2$, and $x_3$ are the input variables.

(modulo all permutations) is much more expensive([5]).

**Definition 1** *Two tables are **permutation equivalent** if one can transformed to the other by permuting the rows and columns.*

**Definition 2** *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, however, 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.

Consider the table in Figure 2. If we order the rows and columns according the number of 1's that appear in each row and column, we obtain the table in Figure 3. 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.

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

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

Figure 2: A simple table annotated with the number of 1's in each row and column.

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

Figure 3: The table of Figure 2 in canonical form

entry $2,3$ would be represented as a vector $(0,0,1,1)$. Summing 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.

These vectors can be transformed to integers to make them easier to manipulate. 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 in a row cannot exceed $m$. By transforming these vectors to base $\max\{m,n\}+1$ 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 columns).

**Definition 3** *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*

$$t_{ij} = \sum_{k=0}^{m_j} (1+\max\{m,n\})^k E_{ij}(k)$$

It is clear that each subset of values at a table entry has unique encoding $t_{ij}$. Figure 4 shows the table of Figure 1 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 4** *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 $\sum_{i=1}^{m} t_{ij} < \sum_{i=1}^{m} t_{ik}$.*

**Definition 5** *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 5 shows the table in Figure 4 converted to semi-canonical form.

$$\begin{array}{cc} & \Sigma = \\ \begin{pmatrix} 1 & 80 & 5 \\ 5 & 4 & 5 \\ 4 & 1 & 4 \end{pmatrix} & \begin{matrix} 86 \\ 14 \\ 9 \end{matrix} \\ \Sigma = \quad 10 \quad 85 \quad 14 & \end{array}$$

Figure 4: The table of Figure 1 converted to a matrix of natural numbers. Row and column sums are also shown.

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

Figure 5: The table of Figure 4 in semi-canonical form

**Theorem 3.1** *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.

# 4 Network Matching

Our aim is, for each node in the new network, to find a node in the old 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.

We cannot use the implementation verification 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 4.3, we show that the problem of finding the best subset can be reduced to finding a maximal prime compatible. In Section 4.4, we present a greedy algorithm for finding a good subset.

## 4.1 Definitions

**Definition 6** *A **network** $N = \{n_i\}$ is a set of nodes 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.*

Figure 6 depicts a typical network. We only consider acyclic networks. Formally, $n \notin \text{tf}(n)$, where $tf(n)$ denotes the set of nodes in the transitive fanin of $n$.

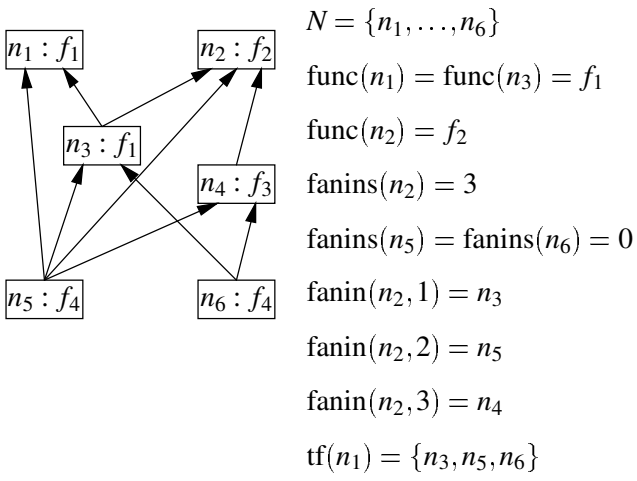


Figure 6: A typical network. The arrows indicate the direction of information flow.

**Definition 7** *The **transitive fanin** of a node n is the set of nodes* $\text{tf}(n) = \cup_{k=1}^{\text{fanins}(n)} (\text{fanin}(n,k) \cup \text{tf}(\text{fanin}(n,k)))$.

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 8** *A pair of nodes* $n_1, n_2$ *is a **candidate pair** (denoted* $n_1 \sim n_2$*) if* $\text{func}(n_1) = \text{func}(n_2)$, $\text{fanins}(n_1) = \text{fanins}(n_2)$, *and* $\forall_{k=1,\ldots,\text{fanins}(n_1)} \text{fanin}(n_1,k) \sim \text{fanin}(n_2,k)$. *Note that the correspondence between the fanins is determined by reducing the table to its semi-canonical form, and noting that in that form, the variable of column i in the table for n* must *correspond with the variable in column i in the table for* $n'$.

Note that this definition implies that all primary inputs may match with each other. We add the caveat that the primary inputs may match provide 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 7 depicts a contradictory situation.

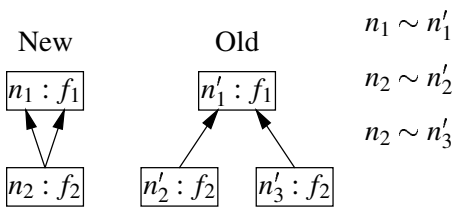


Figure 7: A candidate pair ($n_1 \sim n_1'$) with no consistent matching.

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

**Definition 9** *Given two networks $N$ and $N'$, a **matching** is a function $M : N \to N' \cup \{\phi\}$ such that $M(n) \neq \phi$ implies $n \sim M(n)$ and $\forall k = 1, \ldots, \text{fanins}(n)\ .\ M(\text{fanin}(n,k)) = \text{fanin}(M(n),k)$.*

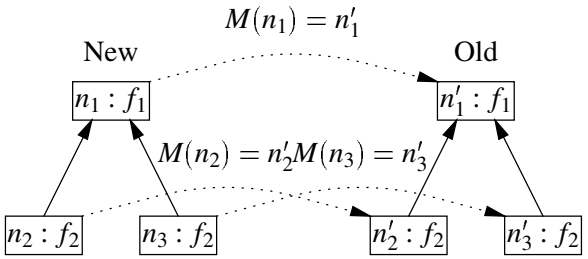


Figure 8: A matching with $\text{q}(M) = 3$

Our objective is to find a matching that maximizes the number of matched nodes (called the quality of the match).

**Definition 10** *The **quality** of a matching $M$ is the number of matched nodes, i.e.,* $\mathrm{q}(M) = |\{n \mid M(n) \neq \phi\}|$.

**Definition 11** *If it exists, the **implied matching** of a candidate pair $n_1 \sim n_2$ is the function*

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

**Theorem 4.1** *An implied matching is a matching.*

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—can not be combined. We need a formal definition of which matchings can be merged:

**Definition 12** *A pair of matchings $M_1$ and $M_2$ are **compatible** (written $M_1 \rightleftharpoons M_2$) if $(M_1(n) \neq \phi) \wedge (M_2(n) \neq \phi) \Rightarrow M_1(n) = M_2(n)$.*

**Lemma 4.2** *Compatibility is transitive, i.e., if $M_1 \rightleftharpoons M_2$ and $M_2 \rightleftharpoons M_3$ then $M_1 \rightleftharpoons M_3$.*

**Definition 13** *The **merge** of two matchings $M_1$ and $M_2$, written $M_1 + M_2$, is the function*

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

**Lemma 4.3** *If $M_1 \rightleftharpoons M_2$, then $M_1 + M_2$ is a matching and $M_1 + M_2 = M_2 + M_1$. Moreover, if $M_2 \rightleftharpoons M_3$, then $(M_1 + M_2) + M_3 = M_1 + (M_2 + M_3)$.*

**Lemma 4.4** *Merging only improves quality, i.e., if* $M_1 \rightleftharpoons M_2$*, then* $\mathrm{q}(M_1), q(M_2) \leq q(M_1 + M_2)$.

## 4.2 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 tables (functions) are matched to be matched. We implement this algorithm with a hash table. Nodes with the same table are put into the same initial "bucket" in the hash table. The canonical form of the table imposes a certain order on the fanins of the node. If two node tables in canonical form are equal, then the fanins node corresponding to column $i$ in the node tables, 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 tables). Thus, if at some iteration, any nodes in the same bucket have corresponding fanins in different buckets, then after that iteration, this nodes get put into different buckets.

This algorithm is similar to the algorithm for the computation of equivalent states in an FSM [10], [1]. After this refinement, all pairs of nodes in a bucket are candidates. The algorithm is shown in Figure 9.

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. We plan to address this as part of future work.

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 9: Identifying Compatible Nodes.

### 4.3 An Exact Formulation

Once we have a set of consistent matchings (Section 4.2), we address the problem of finding a maximum compatible matching exactly.

Lemma 4.4 indicates that merging compatible matchings gives better 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 4.5** *If $M$ is a matching, then it is the sum of a finite number of compatible implied matchings, i.e., $M_1 \rightleftharpoons M_2 \rightleftharpoons \cdots M_k$ and $M = M_1 + M_2 + \cdots + M_k$.*

**Proof** Follows from the definition of matching, implied matching, and Lemmas 4.2 and 4.3. ■

We can define a dominance relation [11] [12] as follows:

**Definition 14** *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 15** *A **prime** matching is one that is not dominated by any other matching.*

**Lemma 4.6** *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 $M_1 \geq M_2, M_1 \rightleftharpoons M_2$ and $M_1 = M_1 + M_2$. Lemma 4.4 implies $\mathrm{q}(M_2) \leq \mathrm{q}(M_1 + M_2)$. However, since $M_1 + M_2 = M_1$, it follows that $\mathrm{q}(M_2) < \mathrm{q}(M_1)$. ■

**Theorem 4.7** *A maximum matching is a prime matching and can be built from a set of compatible implied matchings.*

**Proof** Follows from Lemmas 4.5 and 4.6. ■

Thus, from Theorem 4.7 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 $O(3^n/n)$ [13] and our $n$ 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.

## 4.4 A Greedy 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 4.2 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 10. The performance of our implementation of this algorithm on example circuits is discussed in Section 5.

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
$M(n) = \phi$, the empty matching
**for** $M_i$ largest to $M_i$ smallest
  **if** $M \rightleftharpoons M_i$
    $M = M + M_i$
**return** $M$

Figure 10: The greedy algorithm.

# 5 Results

We have implemented the algorithms described in the VIS [14] environment.

We had to design a set of experiments to test our procedure. 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 and ISCAS benchmark examples and optimized them using a synthesis algorithm ("script.delay" in the SIS [15] system), to obtain a circuit called "new". The original benchmark spec corresponds to the "old" design.

As an experiment we built the function BDDs associated with the "old" design.

| Example | ♯ Nodes Total | ♯ Nodes in Match | Non-Inc Time | Inc Time | Match Initial Time | Match Refine Time | Total Match Time |
|---|---|---|---|---|---|---|---|
| des | 1182 | 1174 | 5.25 | 0.067 | 2.45 | 0.733 | 4.4 |
| i10 | 2754 | 2750 | 21.2 | 0.15 | 0.75 | 1.167 | 3.45 |
| minmax10 | 723 | 87 | 800.734 | 0.2 | 0.133 | 0.2 | 0.35 |
| minmax12 | 914 | 104 | 352.634 | 0.25 | 0.15 | 0.3 | 0.467 |
| mm9a | 682 | 637 | 47.75 | 0.033 | 0.15 | 0.1 | 0.683 |
| mm9b | 714 | 106 | 526 | 0.2 | 0.184 | 0.15 | 0.367 |
| s15850 | 11591 | 10272 | 63.816 | 0.75 | 2.75 | 3.45 | 24.366 |
| clma | 11382 | 10973 | 11.6 | 0.8 | 4.766 | 3.534 | 11.783 |
| clmb | 10842 | 10407 | 11.45 | 0.8 | 4.634 | 3.416 | 10.45 |
| s38584 | 23775 | 20839 | 10.85 | 1.35 | 5.767 | 7.267 | 138.434 |

Table 1: Comparison Incremental and Non_Incremental

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 (Total 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 results on 9 sample examples. Columns 2 and 3 list the total and matched number of nodes in the network respectively. Column 4 lists the time

for a non-incremental BDD computation. The incremental BDD computation time is listed in Column 5. The matching times are listed by its component; i.e. time to get the initial matching(Match Initial Time), time to refine the partition(Match Refine Time), in Column 6 and 7 respectively, as well as the total time to match (Total Match Time) = initial + refine +time to generate and evaluate the quality of the entire matching cones.

The times for incremental BDD computation alone were always better than the non-incremental time (obviously using precomputed information is better than no information). However, when we add in the matching time, this is not always the case. Of the 9 examples reported, 6 had significantly better total times for the incremental procedure (match time + incremental time) as compared to the non-incremental procedure. Two (clma, clmb) examples had almost equivalent times, and one (s38584) had worse time for the incremental method.

We also report the results on the exact computation (Section 4.3)as compared to the heuristic (Section 4.4). 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 2 shows that for examples where the exact method could complete, the heuristic answers were almost always the same.

# 6 Conclusions and Future Work

We have implemented the matching algorithms, and demonstrated that on BDD building the incremental procedures take less time than the non-incremental. We can conclude that on an average the incremental procedure should less time than the non-incremental. We have also implemented the exact matching, and shown that for small examples the exact answer is almost identical to our heuristic. This demonstrates the effectiveness of our heuristic.

| 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 2: Exact Vs 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 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, we plan to use the work of Malik [16] to detect symmetries and 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. We also plan to try to implicitly formulate the matching algorithms, so as to overcome some of the size limitations.

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] 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*, pp. 130–133, Nov. 1990.

[2] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. C-35, pp. 677–691, Aug. 1986.

[3] D. Brand, "Incremental Synthesis," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 126–129, Nov. 1992.

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

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

[6] 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," Tech. Rep. UCB/ERL M91/97, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Nov. 1991.

[7] G. M. Swamy and R. K. Brayton, "Incremental Formal Design Verification," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 458–465, Nov. 1994.

[8] G. M. Swamy, V. Singhal, and R. K. Brayton, "Incremental methods for Fsm Traversal," in *Proc. Intl. Conf. on Computer Design*, pp. 590–595, Oct. 1995.

[9] O. Sokolosky and S. Smolka, "Incremental Model-Checking in Modal Mu-Calculus," in *Proc. of the Conf. on Computer-Aided Verification*, vol. 818, pp. 351–363, Springer-Verlag, June 1994.

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

[11] 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.*, pp. 684–690, June 1994.

[12] G. M. Swamy, P. Mcgeer, and R. K. Brayton, "An Exact Logic minimizer using BDD based Methods ," Tech. Rep. UCB/ERL M92/127, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1992.

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

[14] R. K. Brayton et al., "VIS: A System for Verification and Synthesis," in *Proc. of the Conf. on Computer-Aided Verification*, pp. 332–334, 1996.

[15] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Sequential Circuit Design Using Synthesis and Optimization," in *Proc. Intl. Conf. on Computer Design*, pp. 328–333, Oct. 1992.

[16] 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.