A Semi-Canonical Form for Sequential AIGs

Alan Mishchenko Niklas Een Robert Brayton
Department of EECS, University of California, Berkeley
{alanmi, brayton}@eecs.berkeley.edu niklas@een.se

Michael Case Pankaj Chauhan Nikhil Sharma
Calypto Design Systems
{mcase, pchauhan, nsharma}@calypto.com

Abstract

In numerous EDA flows, complex and time-consuming computations are repeatedly applied to sequential circuits. This situation calls for methods to determine what circuits have been processed already by a tool. This paper proposes an algorithm for semi-canonical labeling of nodes in a sequential AIG. This labeling allows problems or sub-problems solved by an EDA tool to be cached. This can result in a speedup when the tool is applied to designs with isomorphic components or design suites exhibiting substantial structural similarity.

1. Introduction

In sequential and-inverter graphs (AIGs) there frequently exist many instances of the same logic sub-circuit. Many synthesis and verification packages spend significant time analyzing logic, and the presence of identical sub-circuits results in duplication of effort.

Alternatively, if a synthesis or verification package is invoked on a design and then again on a minimally changed version of the same design, many of the logic sub-circuits remain unchanged between runs. Time spent re-analyzing these sub-circuits results in a duplication of effort.

A method for semi-canonical labeling of sequential AIGs proposed in this paper detects the majority of isomorphic subcircuits. Intermediate results computed on these sub-circuits can be cached and applied across every instance of the sub-circuit, leading to dramatic improvements in the runtime of the synthesis or verification package.

A sequential AIG represents a sequential circuit as a directed labeled graph consisting of two-input AND nodes, primary inputs, primary outputs, and flip-flop nodes. The edges are labeled with 1 denoting inversion or 0 denoting no inversion. It is arbitrary how the nodes are labeled and in which order the AND gate fanins are specified.

The idea of this paper is to use this arbitrariness to re-label the nodes based on their transitive fanin and fanout graph structures to create a semi-canonical form for the given sequential AIG. Thus a new labeling of the nodes is derived and each node's fanins are listed in numerical order of their labels. We want the semi-canonical structure to be as precise as reasonably possible. For example, given two isomorphic AIGs, we would like their newly labeled representations to be **identical**. If this always happens, the labeling is canonical. While it is not efficient to make re-labeling exactly precise in this sense, we want it to be highly precise in that it rarely fails to identify two isomorphic AIGs. We call this labeling *semi-canonical*.

Three applications of this idea are as follows.

1. It can detect structurally isomorphic primary outputs (POs) of a multi-output sequential AIG. This is done by taking each output's cone and mapping it into its semi-canonical form. If for two outputs, the forms are identical, then the outputs are isomorphic.
2. In verification where each output represents a property to be proved or disproved, if two outputs are isomorphic then it is only necessary to prove or disprove one. Moreover, if an inductive invariant (or a counter-example) is computed to witness a proof (or a failure) of one property, it can be readily remapped to be a witness for the other. Thus when verifying a set of outputs (properties), only representatives of each class have to be considered, thereby reducing the number of proof obligations.
3. It can be used to cache synthesis or verification results that have been computed previously on an AIG. When an AIG is to be processed, it is cast into its semi-canonical form and stored in a cache along with the computed result. Given a new AIG, we check if it is cached by first computing its semi-canonical form. If it is already cached, we return its saved result. Otherwise, we solve the problem and cache the semi-canonical form along with the result.

We find a re-labeling of the node ids using the AIG structure. This is done by computing signatures for each node, based the node's transitive fanin cone in a kind of virtual unrolling the sequential AIG. We assign unique labels to each node that has a unique signature. If there are nodes that do not have a unique signature, "tie-breaking" is done where one such node is assigned an unused label and node signatures are updated based on this. The key for this semi-canonicalization is a) to define the signature of a node in such a way that it is very precise and b) to compute and update signatures efficiently.

To summarize, the contributions of this paper are:

- An algorithm for structural semi-canonical labeling for sequential AIGs is given. The method of re-labeling can be adapted easily to other forms of graphs.
- Analysis of public and industrial benchmarks confirms that a) re-labeling can be computed efficiently, and b) many benchmarks contain a large number of isomorphic primary output cones.

The rest of the paper is organized as follows. Section 2 describes some background. Section 3 describes the proposed algorithms for semi-canonical labeling. Section 4 reports experimental results. Section 5 concludes the paper and outlines future work.

2. Background

A *sequential AIG* is a graph composed of the following objects: a constant node, primary inputs, flop outputs, flop inputs, primary outputs, and internal AND nodes.

A graph G is a set of nodes $\mathrm{V}(G) = \{1,...,N\}$ and edges $\mathrm{E}(G)$ connecting pairs of nodes. The nodes and edges may have certain attributes. In a sequential AIG, the node attributes are the node type {*PI, PO, FF, internal*} and the edge attributes are {*direction, complementation*}. We say that two graphs are identical ($H \equiv G$) if their representations are identical. This can be done for example by writing the two graphs into files and verifying that the files have identical contents (using *diff*).

For a sequential AIG, the conventional representation [2] is a sequence of triplets $\{(a_1,b_1,c_1),...,(a_N,b_N,c_N)\}$ where a_i,b_i,c_i are node labels, a_i identifying a 2-input AND gate and b_i,c_i its two fanin nodes. b_i,c_i are negated node labels if the corresponding edge contains an invertor. The parent node $\{a_i\}$, must appear in topological order in the list, and each child pair (b_i,c_i) must be in ascending order.

An **isomorphism** *f* between two attributed graphs, *H* and *G*, is a 1-1 mapping *f*: V(*G*) ➔ V(*H*), where nodes $(u, f(u))$ have the same attributes, and edges $(u \to v, f(u) \to f(v))$ have the same attributes. Two graphs are said to be **isomorphic** ($G \approx H$) if there exists an isomorphism between the two. Given a graph *G*, any graph *H*, such that $H \approx G$ can be derived by re-labeling its set of nodes V(*G*) randomly.

Graph canonization is the problem of finding a mapping *canon*: V(*G*) ➔ V(*G*) such that

1. $canon(G) \approx G$, and
2. $[\, canon(H) \equiv canon(G)\,] \Leftrightarrow [\, H \approx G\,]$.

canon(G) is said to be a **canonical form** for G and *canon* is said to be a **canonical mapping** for G.

In this paper, we give an algorithm for creating a mapping (re-labeling of nodes of V(*G*)), *iso,* such that $iso(G) \approx G$. Clearly if $iso(H) \equiv iso(G)$, then $H \approx G$. The hard part is that if $H \approx G$, then $iso(H) \equiv iso(G)$. We do not claim that *iso* is a canonical mapping, but we do claim that for the majority of graphs $H \approx G$, then $iso(H) \equiv iso(G)$. We call such a mapping a **semi-canonical mapping**. Although *iso* is not canonical in general, it might be canonical for certain graph structures.

Canonical label of graphs is a well-studied topic [13]. This paper considers a special case of canonical labeling when graphs are sequential AIGs.

3. Algorithm

3.1 Motivating example

Consider the AIG with inputs {a,b,c,d} and outputs {F, G}, shown on the left of Figure 3.1.

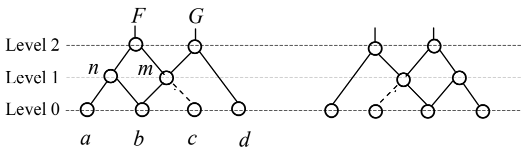

Figure 3.1: Illustration of canonical labeling.

To compute a semi-canonical labeling of the objects in the AIG, we need to distinguish the nodes using their graph properties, independent of their current names.

For example, the circuit has one PI (b) and one internal node (m) with two fanouts. Since these nodes are unique on their levels (level 0 and level 1), they differ from other objects based on their level and fanout count. Next, note that there is only one PI (c) pointed to by a complemented edge. This allows c to be distinguished from other PIs. One of the PO nodes (F) has both fanins on the same level, while the other PO node (G) has fanins on different levels. These observations allow to the POs to be distinguished. Other nodes (a, d, n) can be distinguished similarly.

At the time when a node is distinguished, it can be assigned a unique label. As a result, if the graphs are isomorphic, their semi-canonical labels will be identical with high probability. If they are, then graph isomorphism is easily checked. The check may fail, but in practice using the proposed method of re-labeling, this seldom happens as experiments attest.

For example, consider the AIG shown on the right of Figure 3.1. If we identify its objects using the same rules as we used for the circuit on the left, we will assign semi-canonical labels in the same order. Thus isomorphism is easily established by checking that the resulting representations are identical.

3.2 Computing semi-canonical labels

This section describes the proposed method for computing a re-labeling of the AIG objects to derive a semi-canonical AIG structure. Each node will be given a signature and a label (or new node id). Each edge will be given a value. We begin by defining an edge value which will depend on the edge source node (driver node) label, whether it is complemented or not, and its driver level. Next, we show how node signatures are computed from neighboring node signatures and the edge values. Then an iterative procedure is given to increase the uniqueness of the signatures and to assign labels to nodes as they are uniquely identified by their signatures. Finally, we describe a tie-breaking method that is invoked when two nodes are not distinguised by their signatures.

3.2.1 Computing edge values

An *edge value* captures structural properties of an AIG edge, in particular: (a) the presence of the complemented attribute, (b) the logic level of its source node or sink node, and (c) the fact that source or sink node of the edge are assigned a semi-canonical label. The edge value depends on label assignments through this computation.

```
int getEdgeValue( int edge, int iter ) {
    driver = Node(edge)
    if ( iter == 0 )
        return hash ( Level(driver), Compl(edge) );
    elif ( canonical label of driver has been assigned )
        return hash( Label(driver), Compl(edge) );
    else
        return 0;
}
```

Figure 3.2: Computing edge values.

The edge value is computed using procedure *getEdgeValue*() in Figure 3.2. This procedure takes an edge and the number of the refinement iteration. For the initial iteration (iter = 0), the edge value is computed using two parameters: the level of the source node of the fanin edge, and the complemented attribute of this edge. In other iterations, the level information is not used; instead a node label is used if one has been assigned to the driver node. Otherwise, the edge value is 0.

The first if-statement increases efficiency by increasing the amount of information stored in edge values at the beginning of the computation. The choice of hash function is arbitrary. Using a hash function mapping an interger value into a range of 256 random 32-bit values results in excellent distinguishing power. Function Node() returns the source node of the given incoming edge; Compl() returns 1 if the edge is complemented and 0 otherwise; Level() returns the logic level of a node in the sequential AIG.

We emphasize that when a node has been assigned a label, subsequent edge values will be affected.

3.2.2 Node signature propagation

Initially, all node signatures are set to 0. During *signature propagation*, the AIG is repeatedly traversed in the forward (backward) direction while edge values are added to node signatures. As a result, some nodes (*singleton nodes*) acquire unique signatures.

At the end of each forward (backward) traversal, the newly derived singleton nodes are sorted by their signature value and then assigned labels in that order. These labels impact subsequent edge value computations, as shown in Figure 3.2, and hence subsequent node signature computations. As a result, when a node is visited its updated signature reflects the node labels determined in all previous rounds.

```
void propagateSignaturesForward (
    aig A,          // A is a sequential AIG
    signatures S,   // current signatures of all nodes
    labels U)   // partial assignment of semi-canonical node labels
{
    N = <random number>;
    for each primary input n of aig A
        n->sig = N;  // assign the same random number to all Pis
    for each node or CO object n of aig A (in topological order)
        for each edge e connecting fanin node f with node n {
            e->value = getEdgeValue( e, iter );
            n->sig = (n->sig + f->sig + e->value) mod 2^32;
        }
    for each pair of flop output fo and flop input fi of A
        fo->sig = (fo->sig + fi->sig) mod 2^32;
}
```

Figure 3.3: Propagating node signatures forward.

The procedure for propagating node signatures forward is shown in Figure 3.3. It begins by assigning the same random number to all PIs. Next, it considers all internal nodes and COs in a topological order. For each node or CO, it computes edge values for their fanin edges. Next, the node signature is computed as the sum of (a) its old signature, (b) its node-fanin signatures, and (c) its fanin-edge values. Finally, each flop output node-signature is updated by adding the node signature of its flop input.

The procedure for propagating signatures backward is similar to that shown in Figure 3.3. The differences are:

- starting signatures for primary inputs are not assigned and signatures of the POs are not changed;
- the AIG is traversed in the reverse topological order;
- instead of fanins and fanin edges, fanouts and fanout edges are considered;
- signatures are transferred from flop outputs to flop inputs and not vice versa.

3.3 Refining node signatures

We require each node in the sequential AIG to have a unique signature. *Refinement* refers to the process of updating signatures for every node to reduce the number of nodes that share the same signature.

Node signatures for a sequential AIG are computed as shown in Figure 3.5 using signature propagation.

```
void performRefinement (
    aig A,           // A is a sequential AIG
    signatures S,    // current signatures of all nodes
    labels U )       // partial assignment of labels to nodes
{
    while ( refinement happens )
        propagateSignaturesForward( A, S, U );
        Assign labels to nodes with unique signatures (singletons);
    while ( refinement happens )
        propagateSignaturesBackward( A, S, U );
        Assign labels to nodes with unique signatures (singletons);
}
```

Figure 3.5: Refining signatures.

Signature propagation is performed first in the forward direction and this is iterated while equivalent classes of signatures are changing. It is important to note that node labels are assigned as soon as they are uniquely identified. If non-singleton nodes exist after the forward iteration, then propagation is done in the backward direction starting with the already computed node signatures.

The equivalence classes of nodes, in terms of their signatures, either define isomorphic sub-graphs, or present rare hard cases calling for more elaborate refinement strategies, which are discussed next. If all nodes have been assigned canonical labels at this point, we conjecture that the re-ordering provides a canonical mapping.

The computation of node signatures as sums of edge values is motivated by the need to distinguish nodes based on their structural information, such as the level-by-level distribution of nodes and complemented edges in the transitive fanin/fanout cones of the node.

3.4 Breaking ties

As an example of the need for tie-breaking, consider first the left branch of the AIG shown in Figure 3.4. The transitive fanin cone of node F is composed of two isomorphic groups of AIG nodes (representing XOR gates). Thus the fanins of F would not have unique node labels after signature refining.

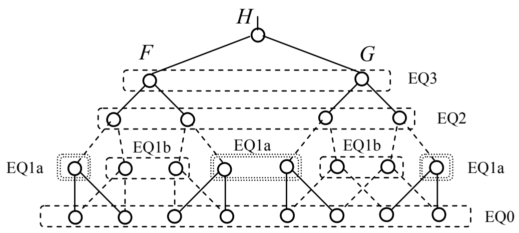

Figure 3.4: Illustration of node equivalence classes.

However, this can happen also even if the graph has does not have isomorphic sub-graphs. Consider the transitive fanin cone of node H in Figure 3.4 (ignoring the fact that F contains isomorphic sub-graphs). The equivalence classes of nodes in this cone are shown in dotted rectangles. In particular, equivalence class EQ3, composed of structurally different nodes F and G, cannot be refined by forward and backward signature propagation. This is because transitive fanin/fanout cones of nodes F and G have the same level-by-level distribution of nodes and complemented edges.

If there are unlabeled nodes after signature propagation, we select one equivalence class and assign its representative the next available label, as shown in Figure 3.6. Then refinement is continued. If there remain unassigned nodes, this is repeated.

```
void **performTieBreaking** (
   aig $A$,          // $A$ is a sequential AIG
   signatures S,     // current signatures of all nodes
   labels U )   // partial assignment of labels to nodes
{
    while ( there are non-trivial equivalence classes ) {
          select one class with the highest logic level;
          assign the next label to a chosen node in this class;
          while ( refinement happens )
               performRefinement( A, S, U );
    }
}
```

Figure 3.6: Breaking ties.

3.5 Main procedure

The self-explanatory top-level algorithm is shown in Figure 3.7.

```
labels deriveSemiCanonicalLabels (
   aig A )           // A is a sequential AIG
{
   labels U;        // node labels
   signatures S;  //node signatures
   // initialize node labels and node signatures
   for each object n of A
         U[n] = 'unassigned';   S[n] = 0;
   // perform initial refinement of labels
   while ( refinement happens )
         performRefinement( A, S, U );
   // perform subsequent refinement while breaking equiv. classes
   while ( unassigned node labels )
         performTieBreaking(A, S, U);
    return U;
}
```

Figure 3.7: Deriving semi-canonical labels.

3.6 Creating the semi-canonical AIGER file and checking isomorphism

Once the semi-canonical labels have been computed, we gather up the nodes in the order of their new labels along with their fanins, ordered according to increasing fanin node label (with inverted edges having their source node id negated). This produces a sequence of N triples which can be written into an AIGER file [2]. The check for isomorphism is done simply by traversing the semi-canonical triplets in order for the two files and comparing triplets. If ever there is a mismatch, the two AIGs are declared non-isomorphic, otherwise they are proved isomorphic because the node labeling provides an isomorphic one-to-one mapping between the two graphs.

3.7 Filtering isomorphic primary outputs

We can detect and remove primary outputs whose sequential logic cones are proved isomorphic to other sequential logic cones of primary outputs in the same AIG. The method is shown in Figure 3.8.

```
aig filterIsomorphicOutputs (
  aig A )      // A is a multi-output sequential AIG
{
   classes C = computeApproxPoEquivClasses( A );
   for each equivalence class c in C {
       for each PO p in equivalence class c {
            aig Ap = extractSequentialLogicCone( A, p );
            labels U = computeSemiCanonicalLabels( Ap );
            compute semi-canonical form F of Ap using U;
            if ( F is differs from that of the representative of c )
                 refine C;
       }
   }
   create set R of one representative POs, for each class in C;
   create a new AIG N consisting of only POs in R.
   return N;
}
```

Figure 3.8: Filtering structurally isomorphic POs.

Computation in Figure 3.8 begins by detecting an over-approximation of equivalence classes of primary outputs using procedure **computeApproxPoEquivClasses()**. This procedure applies only forward signature propagation, which corresponds to quitting after the first while-loop in procedure **performRefinement()** shown in Figure 3.5.

As a result, only the POs, which have different signatures of their transitive fanin cones, are distinguished. For example, Figure 3.9 shows three structurally different AIGs. The first two can be distinguished by the forward procedure, because one of them has a complemented edge (dashed) while the other does not. The last two cannot be distinguished because the node signatures computed using forward propagation are identical in this case, so backward propagation is needed.

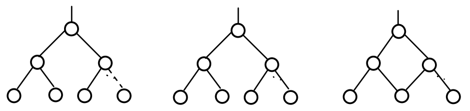

Figure 3.9: Three structurally different AIGs.

After this, if there are non-singular classes, these are checked for isomorphism and if conflicts are found these classes are refined. At the end, we are guaranteed that the remaining POs are truly the representatives of isomorphic equivalence classes.

4. Experimental results

4.1 Public benchmarks

The procedure for filtering isomorphic primary outputs, shown in Figure 3.5, is implemented in ABC [3][4] as a new command *&iso*. This command takes a sequential AIG and reduces it by dropping the POs whose sequential logic cones are isomorphic to those of some other PO (i.e. keeping only the class representatives).

Command &iso can be used, for example, in property verification when a multi-output sequential miter is to be solved. Applying &iso reduces the miter to include only an irredundant set of properties. In this case, if a property is proved UNSAT, all the properties belonging to the same equivalence class are UNSAT. If a counter-example is derived, there exists a mapping which can remap it into a valid counter-example for other properties of the equivalence class. This mapping is currently computed but not returned via the command line.

Results for the largest 8 circuits from the ISCAS benchmark suite are shown in Table 4.1.

The first column shows the benchmark name. The second column shows the number of objects in the sequential AIG. The next three columns compare the number of PO equivalence classes in (a) the original circuit, (b) after partial refinement with forward signature propagation (function *computeApproxPoEquivClasses*), and (c) after applying complete refinement by &iso. Finally, the last two columns show the runtime of forward signature propagation and that of &iso.

Observations:

1. Runtimes are small (2 sec for all benchmarks).
2. There are many isomorphic outputs.
3. Approximate classes are very close to the real isomorphic classes.

4.2 Industrial benchmarks

Table 2 shows industrial benchmarks obtained from IBM. The second column shows the initial AIG size in terms of the number of PIs, POs, FF, and AND nodes.

These designs have many constant POs representing trivial properties. Column 3 shows the number of POs remaining after removal of constant POs.

Next, we run &iso to remove the isomorphic POs. Column 4 shows the number of POs that remain, and Column 5 shows the time to do this. Finally, we perform sequential synthesis and show the number of remaining (unsolved) POs in Column 6. We apply &iso once more to this model, reducing the number of POs as shown in Column 7. The runtime for this &iso call is in Column 8.

Observations:

1. Runtimes of &iso are typically small, except for a few benchmarks. This may be caused by a significant amount of tie-breaking needed.
2. After removing constant outputs, most benchmarks have a significant number of isomorphic outputs.
3. Surprisingly, more isomorphisms were detected after heavy sequential synthesis. The runtime of these detections were always small.

4.3 Measuring the precision of &iso

The algorithm implemented in command &iso cannot be perfect in detecting isomorphisms, because is relies on tie-breaking to resolve ambiguities randomly, while picking noted in a given order, which is not unique.

We did some initial Monte Carlo experiments which showed that in many cases &iso finds all isomorphisms but we found a few where the hit rate was only about 7% -14%.

The Monte Carlo experiment was done as follows:

Choose a non-trivial single-output cone from one of the 21 industrial examples of Table 4.2. This was written out as file1 using ABC command *write_aig -u file1.aig*. During this writing, the -u option causes the AIG to be written into the AIGER file in a canonical order. Next, the original cone is permuted using ABC command *permute*. This randomly re-orders the PIs, POs and FFs of the AIG. Then, it is written using *write_aig -u file2.aig*. Finally *file1* and *file2* are compared using the Unix command *diff*. This was repeated 100 times, and the number of times the two files were the same was recorded.

Although the results for some single-output cones are disappointing, precision is 100% on many benchmarks.

We also experimented on a multi-output benchmarks by applying *permute* followed by *&iso* and comparing the number of isomorphic POs before and after permutation. The numbers always matched and justifies the use of *&iso* in the applications mentioned in this paper.

We conjecture that this discrepancy in precision can be explained by the fact that in a multi-output benchmark, the PIs and FFs are permuted in the same way for all POs. So when each cone is extracted, the set of output cones are 'entangled' and isomorphism detection is thus easier.

In the final version of the paper, we will enhance *&iso* with better tie-breaking methods and a full experiment on precision will be included.

5. Conclusions

This paper describes an efficient algorithm for computing a semi-canonical labeling of nodes in a sequential AIG. The labeling is used to derive a semi-canonical AIG structure, which helps to re-use computed results in EDA tools solving repetitive complex tasks on industrial designs.

Related previous work includes methods for symmetry detection, focused on computing automorphisms of digital circuits [5][6][9][12][17] and functional symmetries [15][18] with applications in BDD variable reordering [16], reachability analysis [8], logic synthesis [14], SAT [1][11], and post-placement rewiring [7], to mention just a few.

To our knowledge, this paper is the first to propose an algorithm for semi-canonical labeling [13] of nodes in sequential AIGs. This might be extended to general directed graphs, where a feedback node cut set plays the role of flip-flops. If the cut set is identified uniquely, then the method of this paper can be applied to identify isomorphisms.

Future work may include:

- Adding incremental propagation of node signatures when the number of updated equivalence classes is relatively small, using an approach similar to [10]. This might speed up the computations for very large AIGs.
- Generalizing the algorithm to work for logic networks other than the traditional AIGs. For example, expanding two-input AND nodes into multi-input AND nodes may prove two circuits isomorphic, even if the original AIGs had different two-input node structures.

Acknowledgements

This work was partly supported by SRC contract 1875.001 and NSA grant "Enhanced equivalence checking in crypto-analytic applications". We also thank industrial sponsors of BVSRC: Altera, Atrenta, Cadence, Calypto, IBM, Intel, Jasper, Magma, Microsemi, Real Intent, Synopsys, Tabula, and Verific for their continued support.

6. References

[1] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Solving difficult SAT instances in the presence of symmetry", *IEEE TCAD'03*, vol. 22(9), pp. 1117-1137.

[2] A. Biere, *AIGER format*. http://fmv.jku.at/aiger/

[3] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[4] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool", *Proc. CAV'10*, LNCS 6174, pp. 24-40.

[5] D. Chai and A. Kuehlmann, "A compositional approach to symmetry detection in circuits", *Proc. IWLS'06*, pp. 228-234.

[6] D. Chai and A. Kuehlmann, "Symmetry detection for large multi-output functions", *Proc. IWLS'07*, pp. 305-311.

[7] K.-H. Chang, I. L. Markov and V. Bertacco, "Post-placement rewiring by exhaustive search for functional symmetries", *ACM TODAES*, vol. 12(3), #32, August 2007.

[8] P. Chauhan, P. Dasgupta, and P. P. Chakraborty, "Exploiting graph isomorphism for BDD compaction and faster simulation", *Proc. VLSI'99*, pp. 224-229.

[9] P. T. Darga, K. A. Sakallah, and I. L. Markov. "Faster symmetry discovery using sparsity of symmetries". *Proc.DAC'08*, pp.149-154.

[10] N. Een and A. Biere, "Effective preprocessing in SAT through variable and clause elimination", *Proc. SAT'05*.

[11] H. Katebi, K. A. Sakallah, and I. L. Markov, ``Symmetry and satisfiability: An update", *Proc. SAT'10*, pp. 113-127.

[12] H. Katebi, K. A. Sakallah, and I. L. Markov, "Conflict anticipation in the search for graph automorphisms", *Proc. Int'l Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, Merida, Venezuela, 2012.

[13] B. D. McKay, "Computing automorphisms and canonical labeling of graphs", *Combinatorial Mathematics, Lecture Notes in Mathematics*, 686, pp. 223-232 (Springer-Verlag, Berlin, 1978).

[14] V. N. Kravets and K. A. Sakallah, "Constructive library-aware synthesis using symmetries". *Proc. DATE'00*, pp. 208-213.

[15] A. Mishchenko, "Fast computation of symmetries in Boolean functions", *IEEE TCAD'03*, vol. 22(11), pp.1588-1593.

[16] S. Panda, F. Somenzi, and B. Plessier, "Symmetry detection and dynamic variable ordering of decision diagrams". *Proc. ICCAD'94*, pp. 628-631.

[17] G. Wang, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "Structural detection of symmetries in Boolean functions", *Proc. ICCD'03*, pp. 498-503.

[18] J. S. Zhang, A. Mishchenko, R. Brayton, and M. Chrzanowska-Jeske, "Symmetry detection for large boolean functions using circuit representation, simulation, and satisfiability", *Proc. DAC '06*, pp. 510-515.

Table 4.1: Computing canonical structure for the largest ISCAS benchmarks.

Example	AIG	POs	POs filter	POs canon	Time filter	Time canon
s13207	2728	121	88	89	0.01	0.11
s13207_1	2728	152	89	89	0.01	0.05
s15850	3526	87	43	43	0.01	0.05
s15850_1	3526	150	47	47	0.01	0.05
s35932	11948	320	320	320	0.01	0.02
s38417	9238	106	37	39	0.01	0.05
s38584	12310	278	218	218	0.01	0.53
s38584_1	12310	304	219	219	0.01	0.53

Table 4.2: Industrial benchmarks.

Example	initial AIG sizes PIs/POs/FF/ANDs	remove const POs	after iso	iso (sec.)	POs after synthesis	after iso	iso(sec.)
SDCU	1245/838/2442/14418	834	727	3.2	361	353	.09
TPC_P	1619/1270/4777/30762	948	795	16.00	394	393	.28
TCP_Oh	1522/626/3838/36890	598	553	4.78	545	541	.55
PC_T	719/278/2565/20101	274	258	.55	129	129	.01
SDXIA	2164/604/7822/78858	600	490	32.04	245	245	.63
DU	3978/5675/10864/84397	946	518	198.14	458	432	.80
VS_V	535/337/15753/126236	337	262	.77	141	139	.13
PCIEX	1138/170/950/25034	168	40	.02	14	14	.01
sdcu_ex	415/514/892/7712	514	514	.08	167	167	.03
GXFM	893/244/2657/18140	240	223	1.04	214	210	.04
IFFBC	759/710/4758/31743	706	639	4.08	313	310	.10
L2DAC	465/294/1277/7743	282	203	1.09	105	101	.01
PB_E7	689/438/2065/15007	434	355	1.06	311	311	.13
PC_TI	892/109/2426/22364	105	64	2.77	0	na	na
xgma_f	322/952/1566/49869	819	804	3.10	0	na	na
jppu	8716/38096/87450/855493	25861	10124	1436.27	9811	9673	7.78
SDXIA	2164/604/7822/78858	600	490	37.02	245	245	.67
MCA_S	1916/732/4620/33448	726	663	20.28	662	662	.03
TPC_Oc	1364/534/5005/45938	524	500	12.85	497	497	3.57
IODMR	583/408/3117/29040	395	388	2.51	387	386	.24