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, time-consuming computations are repeatedly applied to sequential circuits. This motivates developing 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, allowing problems or sub-problems solved by an EDA tool to be cached with their computed results. This can speed up the tool when 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 but with different inputs. 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.

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

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 otherwise. Labeling of nodes and AND gate fanins can be arbitrary.

The idea of the method is to use this freedom, to re-label the nodes based on their transitive fanin and fanout graph structures, and thereby create a semi-canonical form for the given sequential AIG. Thus a new labeling 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 want 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 the forms of the two outputs, are identical, then the outputs are isomorphic.
2. In verification, where each output represents a property to be proved, if two outputs are isomorphic then it is only necessary to solve one. 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, only one representative of each isomorphic class need be considered, 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 compute its semi-canonical form and check if it is cached. If already cached, we return the saved result. Otherwise, we solve the problem and cache the semi-canonical form and the result.

The method finds 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 the sequential AIG. Unique labels are assigned to a node that has a unique signature. For nodes with non-unique signatures, "tie-breaking" is done by assigning one such node an unused label and then 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 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 and Section 5 concludes and outlines future work.

978-3-9815370-0-0/DATE13/©2013 EDAA

2. Background

A *sequential AIG* has a constant node, primary inputs and outputs, flop inputs and outputs, and internal AND nodes. It has node attributes {*PI, PO, FF, internal*} and the edge attributes {*direction, complementation*}. Two graphs are identical ($H \equiv G$) if their representations are identical. Checking this can be done by writing the two graphs into files and verifying their contents to be identical (using *diff*).

The conventional representation of an AIG [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 **isomorphic** ($G \approx H$) if an isomorphism exists between the two. Given *G*, $H \approx G$ 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.

We give an algorithm for re-labeling of nodes of V(*G*)), *iso,* such that $iso(G) \approx G$. Clearly if $iso(H) \equiv iso(G)$, then $H \approx G$. We do not claim that *iso* is a canonical mapping, but we do claim that for the majority of graphs if $H \approx G$, then $iso(H) \equiv iso(G)$. We call such a mapping a **semi-canonical mapping**.

Although canonical labeling is well-studied [14], this paper considers the special case of sequential AIG graphs.

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. A dotted edge denotes inversion.

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.

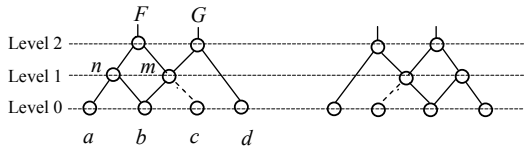

Figure 3.1: Illustration of canonical labeling.

For example, the above 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, which distinguishes it 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 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, this seldom happens, using the method proposed in this paper for re-labeling, 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 the resulting representations to be identical (one is the mirror image of the other).

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 distinguished by their signatures. If tie-breaking is never invoked then the method is precise.

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. Thus, the edge value depends on node label assignments through this computation.

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 replaced by a node label

if one has been assigned to the driver node. Otherwise, the edge value is 0.

```
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 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.

```
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.

At the end of each forward (backward) traversal, the newly derived singleton nodes are sorted by their signature and 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.

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 having the same signature.

Node signatures for a sequential AIG are computed using signature propagation, as shown in Figure 3.4.

```
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 3.4: Refining node 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 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 3.5. 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.

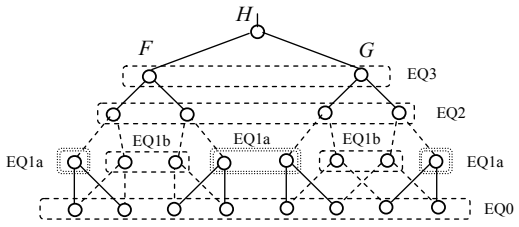

Figure 3.5: Illustration of node equivalence classes.

```
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.

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 3.5 (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 do a heuristic refinement by selecting one equivalence class and assigning its representative the next available label, as shown in Figure 3.6. This effect is propagated to distinguish other nodes. This refinement is continued until there are no unassigned nodes.

3.5 Main procedure

The top-level algorithm is shown in Figure 3.7.

3.6 Creating the semi-canonical AIGER file and checking isomorphism

Once the semi-canonical labels have been computed, we sort 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.

```
labels deriveSemiCanonicalLabels ( aig A )
{
   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 refinement while breaking equivalence classes
   while ( unassigned node labels )
        performTieBreaking(A, S, U);
    return U;
}
```

Figure 3.7: Deriving semi-canonical node labels.

3.7 Filtering isomorphic primary outputs

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

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 3.4.

```
aig filterIsomorphicOutputs ( aig A )
{
   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.

As a result, only the POs with 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 after forward propagation are identical in this case, so backward propagation is needed.

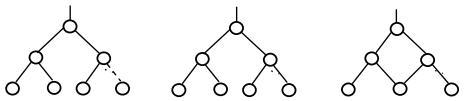

Figure 3.9: Three structurally different AIGs.

After this, if there are non-singular classes, these are checked for isomorphism and, if conflicts are found, these are refined. At the end, 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 3.7, is implemented as command *&iso* in ABC [3][4]. It takes a sequential AIG and removes the POs whose sequential logic cones are isomorphic to those of some other PO (i.e. keeps only the class representatives).

Command *&iso* can be used, for example, in property verification when the problem is a multi-output sequential miter. Applying &iso reduces the miter to include only an irredundant set of properties. If a property is proved UNSAT, all the properties belonging to the same class are UNSAT. If a counter-example is derived, it can be mapped into counter-examples for all properties in the same class.

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 AND nodes in the AIG. The next three columns compare the number of PO equivalence classes in (a) the original circuit, (b) after partial refinement with procedure *computeApproxPoEquivClasses* performing only forward signature propagation and (c) after complete refinement by *&iso*. Finally, the last two columns show the runtime of forward signature propagation and that of *&iso*. This experiment was performed using IntelCore 2Quad Q9450, 2.66GHz. Only one core was used.

Observations from Table 4.1:

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

4.2 Industrial benchmarks

In the second experiment, shown in Table 4.2, command *&iso* was applied to a suite of 19 multi-output industrial verification benchmarks. Each benchmark was converted into a set of single-output problems using ABC command sequence "*cone –s –O <num>; scl*" and given to '*pdr*' [11], the most powerful single-engine prover among those currently available in ABC.

The first column of Table 4.2 shows the number of a verification problem. The following first section of the table shows benchmark statistics before applying *&iso* while the second section shows statistics after applying *&iso*. Inside each section, columns labeled "AND", "FF", "PI", and "PO" show the number AND nodes, flip-flops, primary inputs and primary outputs. The columns labeled "pdr" show the number of POs solved by 'pdr' within the resource limits (5 minutes and 1GB). Columns labeled "time" show the cumulative runtime of 'pdr' on those POs that were successfully solved. Finally, the last column shows the runtime of *&iso* for each benchmark. The bottom row shows geometric averages of the corresponding columns.

This experiment was performed on a cluster of machines running Linux 2.6 on a 2.4 GHz AMD Opteron processor.

Observations from Table 4.2:

1. The number of POs is reduced almost 3x.
2. The number of ANDs and flops is reduced 40%.
3. The total verification time is reduced about 2x, while *&iso* takes only 15% of the total runtime.

4.3 Measuring the precision of &iso

The algorithm implemented in command &iso cannot be perfect, because is relies on tie-breaking to resolve ambiguities randomly, but choosing nodes in a given order.

Initial Monte Carlo experiments 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 experiments were done by choosing a non-trivial single-output cone from one of the 21 industrial examples of Table 4.2. This was written out as *file1.aig* During this writing, the *-u* option causes the AIG to be written 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 the *-u* option as *file2.aig*. Finally *file1* and *file2* are compared for being identical using *diff*. This was repeated 100 times, recording when the two files were the same.

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

We also experimented on multi-output benchmarks by applying *permute* followed by *&iso* and comparing the number of isomorphic POs before and after permutation. The numbers always matched. We conjecture that this discrepancy is because 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 more precise.

5. Conclusions

Related previous work includes methods for symmetry detection, focused on computing automorphisms of digital circuits [5][6][9][13][18] and functional symmetries [16][19] with applications in BDD variable reordering [17], reachability analysis [8], logic synthesis [15], SAT [1][12], and post-placement rewiring [7], to mention just a few.

We believe this is the first paper to propose an algorithm for semi-canonical labeling [14] of sequential AIGs. This method 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.

The proposed method and its public implementation in ABC can be used to avoid repeated time-consuming computations in industrial synthesis and verification tools.

Acknowledgements

This work is 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, Mentor Graphics, 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.. 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] N. Een, A. Mishchenko and R. Brayton, "Efficient implementation of property-directed reachability", *Proc. FMCAD'11*.

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

[13] 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.

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

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

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

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

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

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

Table 4.1: Computing canonical structure for 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: Reductions on industrial verification benchmarks and impact on cumulative verification runtime.

	Industrial circuits in their original form						Industrial circuits after &iso is applied						&iso
Ex	AND	FF	PI	PO	PDR	time, s	AND	FF	PI	PO	PDR	time, s	time, s
0	3307	64	54	40	40	38.8	3177	64	54	20	20	36.8	0.01
1	25478	184	138	184	136	15.8	19493	65	138	65	17	1.9	0.05
2	248642	1291	767	31	0	0.1	168090	951	767	21	0	0.1	1.76
3	1037	56	5	18	0	0.1	970	52	5	9	0	0.1	0.02
4	217292	2483	2277	1203	34	3.4	217202	2483	2277	1188	34	3.4	215.71
5	58432	6	3464	2928	2928	661.9	36745	6	3464	1654	1654	373.0	19.46
6	85926	4156	6510	907	907	90.7	85926	4156	6510	904	904	90.4	15.23
7	162357	331	1019	24	12	1.2	162269	331	1019	13	1	0.1	0.07
8	217292	2483	2277	1203	34	3.8	217202	2483	2277	1188	34	3.8	738.76
9	259731	10993	2362	181	73	3430.2	255978	10859	2362	164	64	3429.3	7.05
10	22325	1289	492	165	165	16.5	11650	596	492	60	60	6.0	0.14
11	180873	7129	14046	12938	0	0.1	92358	2791	14046	4702	0	0.1	28.70
12	94912	8298	4210	32	32	748.9	3307	269	4210	1	1	20.6	0.15
13	212389	229	1363	128	118	95.5	211514	229	1363	27	17	81.8	0.60
14	1009803	66962	125	28	0	0.1	135545	8624	125	21	0	0.1	1.41
15	74480	2976	394	327	0	0.1	18765	900	394	14	0	0.1	4.16
16	63855	6806	1519	413	405	226.9	63149	6805	1519	60	52	161.2	0.40
17	10130	64	33	16	2	0.2	9948	64	33	6	1	0.1	0.05
18	162357	331	1019	24	12	2.1	162269	331	1019	13	1	0.2	0.11
Geo	1.000	1.000	1.000	1.000	1.000	1.000	0.611	0.597	1.000	0.385	0.396	0.495	0.161