# Reviewing AIG Equivalence Checking Approaches

Marcos Backes, Jody Maick Matos,
Renato Ribas and Andre Reis
UFRGS, Instituto de Informatica
Av. Bento Goncalves, 9500
Porto Alegre, RS, Brazil
{mhbackes,jody.matos,rpribas,andre.reis}@inf.ufrgs.br

## ABSTRACT

This work presents a review of and-inverter graph (AIG) equivalence checking approaches. It is well known that AIGs are not canonical structures, i.e., the same functionality can be represented with a different number of AIGs. In this sense, a verification step for checking the equivalence between AIGs is mandatory when performing AIG transformation algorithms, such as rewriting procedures. The work presented herein reviews two main approaches to perform AIG equivalence checking: (1) using BDDs to verify the aimed equivalence, exploring the canonicity of this data structure; and (2) transforming the AIG equivalence checking in a SAT problem, using a SAT solver to perform the checking. The SAT-based equivalence checking adopted in the work presented herein is based on Conjunctive Normal Form (CNF). The CNFs are obtained by using the Tseitin Transformation. Applying the methodology proposed herein, the results show that almost 67% of the benchmark AIGs could not be checked using the BDD approach due to the running time exceeded the threshold of 27 hours. In turn, the SAT approach successfully verified 11 of the 12 benchmark AIGs (almost 67% of them in less the 1 second). The SAT-based approach also achieved a lower memory usage than the BDD-based approach.

## Keywords

Logic synthesis and verification, AIG equivalence checking, BDD, SAT.

## 1. INTRODUCTION

Optimization of multi-level logic networks using logic synthesis plays an important role in automated design flow, specially in cell-based VLSI designs. Logic synthesis is often applied to the network derived by compiling HDLs, such as VHDL or Verilog, and performing both technology-independent and technology-dependent optimizations. The synthesis task performed in this step of integrated circuit design flow defines the logic used to implement a design [9, 11].

The most recent logic synthesis works are based on a type of data structure called and-inverter graphs (AIG) [2, 16]. The AIG data structure (explained in detail further) can be viewed as a graph composed of 2-input AND (AND2) nodes, connected by direct or negated edges [16]. The state-of-the-art logic synthesis tools, such as ABC [2], are able to minimize the number of (AND2) nodes in AIGs. This number can be directly correlated to the number of gates in implementations using 2-input simple gates, such as ANDs, ORs, NANDs and NORs. These gates are variants of the primitive AND2, obtained by applying De Morgan's law and inversions.

Traditional AIG transformation approaches are mainly focused in two specific goals: (1) minimizing the number of AIG nodes, such as the *refactor* [4, 16] and *rewrite* approaches [16]; and (2) optimizing the delay estimation of AIGs, such as the *balance* method [6, 16]. However, AIGs are not canonical structures, i.e., the same functionality can be represented with a different number of AIGs. In this sense, a verification step for checking the equivalence between the AIG before the transformation and the resulting AIG after the transformation is mandatory.

To verify the equivalence between AIGs, the most naive approach is to compare them using a canonical structure. The usual structure for this comparison is the binary decision diagram (BDD) [15]. The BDD (explained in detail further) represents a set of binary-valued decisions, culminating in an overall decision that could be *TRUE* or *FALSE* [1, 7, 11, 13]. Since the reduced ordered BDD (ROBDD) is canonical, the resulting ROBDD for equivalent AIGs shall be exactly the same. Nonetheless, it is possible to perform this equivalence checking using different approaches, such as Boolean satisfiability (SAT). The SAT is known as the problem of analyzing if the variables of a given Boolean function can be assigned in such a way as to make the function evaluate to *TRUE.* A wide range of problems can be transformed into instances of SAT. Algorithms called SAT solvers can efficiently solve subsets of SAT instances, what is true for AIG equivalence checking.

This work reviews the two main approaches to perform AIG equivalence checking found on the literature: (1) using BDDs to verify the aimed equivalence, exploring the canonicity of this data structure; and (2) transforming the AIG equivalence checking in a SAT problem, using a SAT solver to perform the checking. Applying the methodology proposed herein, the results show that the SAT-based approach is more suitable and scalable than the BDD-based approach.

The rest of the paper is organized as follows. Section

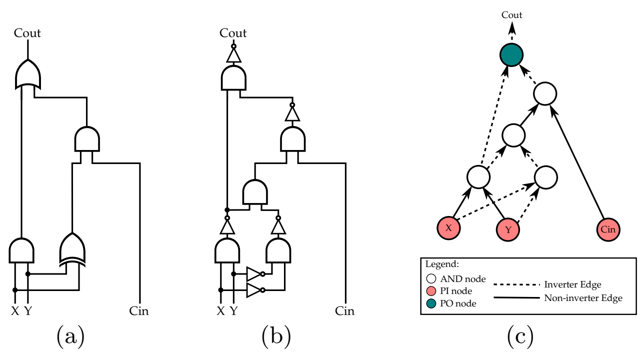

**Figure 1: A combinational circuit (a), its representation with AND2 gates and inverters (b), and one possible AIG representation (c).**

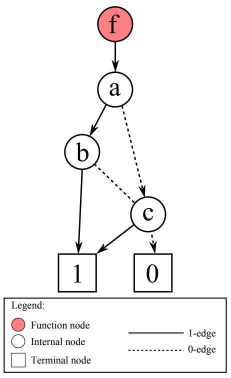

**Figure 3: Example of BDD representing the function $f = ab + c$.**

2 presents a background of AIGs, BDDs, Tseitin Transformation and Boolean Satisfiability (SAT). In Section 3 and Section 4, we present methodologies to compare AIGs using BDDs and SAT, respectively. Section 5 presents the experimental results. Section 6 outlines the conclusions.

## 2. BACKGROUND

This section provides a basic technical background which is essential for understanding the methodology approached by this work: AIGs and BDDs as data structures commonly used on logic synthesis; and the Boolean satisfiability problem (SAT), since SAT solvers can also be used to verify AIG equivalences. The SAT-based equivalence checking adopted in the work presented herein (presented in Section 4) is based on Conjunctive Normal Form (CNF). Due to this reason, we also review the Tseitin Transformation as an efficient way to generate a CNF from a logic network.

### 2.1 And-Inverter Graph

An *And-Inverter Graph* (AIG) is a data structure used in current state-of-the-art logic synthesis tools, like ABC [2]. AIGs are directed acyclic graphs (DAG) with specific types of nodes: 2-input AND (AND2) nodes, primary input (PI) nodes, and primary output (PO) nodes. Primary input nodes have no incoming edges. AND2 nodes have two incoming edges. Any node of an AIG can be labeled as an output node [16]. The edges of an AIG have a specific property: they are either in their positive or complemented form. A Boolean signal arriving at the target node via a positive edge has the same polarity as the source node. The complemented form of an AIG edge indicates the Boolean inversion operation of its signal [16]. Figure 1 shows a possible AIG of the logic function $Cout = ((x \bigotimes y) \cdot cin) + (x \cdot y)$.

AIG is not a canonical structure, i.e., the same function can be represented with a different number of AIGs. Figure 2 shows an example of AIG non-canonicity. In this sense, the task of verifying the Boolean equivalence between two AIGs plays an important role handling this data structure, such as AIG rewriting approaches. In the following, reviews of BDDs, Tseitin Transformation and SAT are presented focusing on AIG equivalence checking.

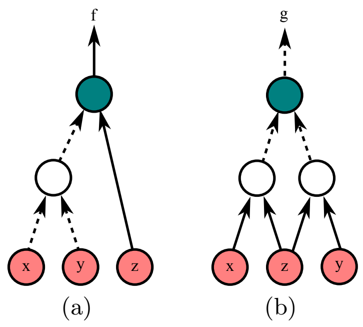

**Figure 2: Example of AIG non-canonicity: (a) AIG representing the function $f = (x + y)z$; (b) AIG representing the function $g = xz + zy$. Notice that $f \equiv g$.**

### 2.2 Binary Decision Diagram

A *Binary Decision Diagram* (BDD) is a graph representation of Boolean functions. In this sense, a BDD represents a set of binary-valued decisions, culminating in an overall decision that could be *TRUE* or *FALSE*. Though BDDs are relatively old [1,13], they just began attracting the research community attention with the work of Bryant [5], which brought out the advantages the Reduced Ordered BDD (ROBDD) as canonical representations [7,11]. For simplicity, we will refer to ROBDDs as BDDs.

Formally, a BDD is a directed acyclic graph (i.e., a directed graph with no directed cycles) with two terminal nodes, called *1-terminal* and *0-terminal*, which denote the *TRUE* and *FALSE* decision, respectively. The nodes of a BDD are partitioned into three subsets: function nodes $\Phi$, internal nodes $V$, and the terminal nodes $\{0, 1\}$. A function node $\phi \in \Phi$ denotes the function being represented, has one outgoing edge and have no incoming edges. Each internal node $v \in V$ has a label $l(v) \in S_F$, where $S_F$ denotes the *support* of a function $F$, i.e., each label represents a variable on which $F$ actually depends. The internal nodes have two outgoing edges: the *0-edge*, which denotes the *FALSE* decision with respect to source node of the edge; and the *1-edge*, which denotes the *TRUE* decision with respect to source node of the edge [7,9,11]. Figure 3 depicts the BDD representing the function $f = ab + c$.

### 2.3 Tseitin Transformation

The Tseitin Transformation is a procedure to generate a Boolean equation in Conjunctive Normal Form (CNF) from

**Table 1: Tseitin Transformation for Simple Gates**

| Gate | Function | Resulting CNF |
|---|---|---|
| NOT | $o = \overline{x}$ | $(x+o)(\overline{x}+\overline{o})$ |
| AND2 | $o = x \cdot y$ | $(\overline{x}+\overline{y}+o)(x+\overline{o})(y+\overline{o})$ |
| OR2 | $o = x + y$ | $(x+y+\overline{o})(\overline{x}+o)(\overline{y}+o)$ |
| NAND2 | $o = \overline{(x \cdot y)}$ | $(\overline{x}+\overline{y}+\overline{o})(x+o)(y+o)$ |
| NOR2 | $o = \overline{(x + y)}$ | $(x+y+o)(\overline{x}+\overline{o})(\overline{y}+\overline{o})$ |
| XOR2 | $o = x \oplus y$ | $(\overline{x}+\overline{y}+\overline{o})(x+y+\overline{o})$ $(x+\overline{y}+o)(\overline{x}+y+o)$ |

a logic network [17]. A CNF formula is a way to represent Boolean equations using only a conjunction of disjunctive clauses, i.e., a Product-of-Sums, or POS. Possible results for applying the Tseitin Transformation for simple gates are shown in Table 1. In order to apply this transformation to more complex circuits, a simple approach is generating the conjunction of the Tseitin Transformation for each simple gate of the circuit. As an example, the Tseitin Transformation of the logic network presented in Figure 4 results in the CNF shown in Equation 1.

$$(\overline{x}+\overline{z}+a)(x+\overline{a})(z+\overline{a})(\overline{z}+\overline{y}+b)(z+\overline{b})(y+\overline{b})(a+b+\overline{o})(\overline{a}+o)(\overline{b}+o) \tag{1}$$

## 2.4 Boolean Satisfiability Problem

The Boolean Satisfiability (SAT) is a problem which, given a Boolean function $f(x_0, x_1, ..., x_n)$, tries to determine if there is a combination of values assigned to the input variables of $f$ which evaluates the function to *TRUE*. If such combination exists, then the function is called *satisfiable*, else, it means that $f = \mathit{FALSE}$ for every possible input and it is called *unsatisfiable*.

SAT was the first known NP-complete problem. Therefore, it is believed (but not proven) that there is no algorithm which efficiently solves *all* SAT problems. A class of algorithms called SAT solvers [8, 12, 14] are capable of solving very efficiently a subset of SAT instances. Most of them receive as input the CNF formula generated by the application of Tseitin Transformation.

$f(x, y) = xy$ is an example of *satisfiable* Boolean function because when $x = \mathit{TRUE}$ and $y = \mathit{TRUE}$, $f(x, y) = \mathit{TRUE}$. An example of unsatisfiable function is $g(x) = x\overline{x}$, which for both possible values of $x$, $g(x) = \mathit{FALSE}$.

## 3. AIG COMPARING USING BDD

For checking the equivalence of AIGs $X$ and $Y$, we create a BDD for each output of both $X$ and $Y$. In this way, we explore the canonicity of BDDs and the advantages of a dynamic programming implementation. If the generated BDDs are the same for every output $x$ in $X$ and its correspondent output $y$ in $Y$, then the AIGs are equivalent.

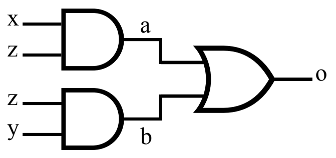

**Figure 4: Circuit representation of $f(x, y) = xz + zy$**

**Table 2: AND2 and INV operations described using ITE.**

| Name | Expression | ITE |
|---|---|---|
| $AND(f, g)$ | $f \cdot g$ | $ite(f, g, 0)$ |
| $NOT(f)$ | $\overline{f}$ | $ite(f, 0, 1)$ |

To create the BDDs from the AIGs, we propose to use the If-Then-Else (ITE) operator [3]. The ITE is a ternary operator which forms the core of recursion based synthesis operations for BDDs. ITE can be viewed as a Boolean function defined for three operands as follows:

$$ite(F, G, H) = F \cdot G + \overline{F} \cdot H \tag{2}$$

It is well known that ITE can be used to implement all two-variable Boolean functions. In this sense, considering that AIGs represent the circuit by using only 2-input ANDs (AND2) and inverters (INV), the ITE operator can be efficiently used to recursively create a BDD from an AIG applying the ITE assignment for each needed operation. Table 2 present both AND2 and INV operations described using the ITE operator.

A big problem when comparing AIGs using BDD approach is that the number of nodes of the generated BDD grows very fast. In the worst case, for $n$ inputs, the size of a BDD is $2^n + 1$ (considering both terminal nodes). Therefore, the size of a BDD increases exponentially as the size of input grows. This fact may cause low time performance and high memory consumption when using BDDs.

## 4. AIG COMPARING USING SAT

The equivalence of Boolean functions can be checked using SAT [10]. Given two Boolean functions $f(x_0, ..., x_n)$ and $g(x_0, ..., x_n)$, to check their equivalence with a SAT-based approach, we first create a new function $h(x_0, ..., x_n) = f(x_0, ..., x_n) \oplus g(x_0, ..., x_n)$. Note that if $f$ and $g$ are equivalent, then $h = \mathit{FALSE}$ for any possible variable assignment, i.e., the function $h$ is unsatisfiable. We generate a CNF from $h$ using Tseitin transformation and apply a SAT solver on this CNF. If $h$ is unsatisfiable then $f$ and $g$ are equivalent, else they represent different Boolean functions.

The method to compare AIGs with SAT used in this work is very alike. Let $X$ and $Y$ be AIGs. To compare their equivalence, we create a new AIG $Z = X \oplus Y$. Considering every node of $Z$ as an AND2 gate and every inverter-edge as a NOT gate, we derive the CNF formula using Tseitin transformation and use this formula as input of a SAT solver. $X$ and $Y$ are equivalent if and only if $Z$ is *unsatisfiable*.

## 5. RESULTS

Both BDD-based and SAT-based methods for comparing AIGs (presented in Section 3 and Section 4) were applied over a set of benchmark circuits. For each of these circuits, an initial AIG was obtained. Then, this AIG was rewritten using *refactor* [4], *balance* [6] and *rewriting* [16] algorithms, resulting in the final AIG. The equivalence checking was done between the final and the initial AIGs using both BDD and SAT approaches. To evaluate the efficiency of the comparison methods, a running time measure was performed.

**Table 3: Runtime analysis for both BDD and SAT AIG equivalence checking when comparing the benchmark circuit to the same implementation after applying rewriting algorithms.**

| CIRCUIT | #IN | #OUT | #NODES | TIME | | MEMORY | |
|---|---|---|---|---|---|---|---|
| | | | | SAT | BDD | SAT | BDD |
| C17 | 5 | 2 | 6 | 14 *ms* | 1 *ms* | 93.3 *MB* | 3.2 *MB* |
| C432 | 36 | 7 | 127 | 58 *ms* | 29.6 *min* | 94.4 *MB* | 3.7 *MB* |
| C499 | 41 | 32 | 386 | 367 *ms* | > 27 *hours* | 99.4 *MB* | 2.3 *MB* |
| C880 | 60 | 26 | 306 | 84 *ms* | 1 *hour* | 94.4 *MB* | 65.7 *MB* |
| C1355 | 41 | 32 | 390 | 777 *ms* | > 27 *hours* | 98.3 *MB* | 2.2 *MB* |
| C1908 | 33 | 25 | 354 | 490 *ms* | 25.4 *hours* | 98.4 *MB* | 44.1 *MB* |
| C2670 | 233 | 139 | 534 | 351 *ms* | > 27 *hours* | 101.3 *MB* | 85.6 *MB* |
| C3540 | 50 | 22 | 918 | 4 *sec* | > 27 *hours* | 110.3 *MB* | 62.3 *MB* |
| C5315 | 178 | 123 | 1323 | 1.3 *sec* | > 27 *hours* | 113.4 *MB* | 8.4 *MB* |
| C6288 | 32 | 32 | 1870 | > 27 *hours* | > 27 *hours* | 1.1 *GB* | 2.6 *GB* |
| C7552 | 207 | 108 | 1377 | 701 *ms* | > 27 *hours* | 132.4 *MB* | 42.3 *MB* |
| i10 | 257 | 224 | 1799 | 1.5 *sec* | > 27 *hours* | 128.4 *MB* | 102 *MB* |
| RATIO | - | - | - | 1.00 | 8.96 | 1.00 | 1.35 |

The proposed approaches were implemented using C++ programing language and compiled with *g++* 4.8.2 compiler. The experiments were performed in a machine with Intel(R) Core(TM) i3-M330 @ 2.33GHz CPU, 4Gb RAM.

Table 3 presents the main results we obtained. Columns labeled *"#IN"*, *"#OUT"* and *"#NODES"* present, respectively, the number of inputs, outputs and AND nodes of the benchmark AIGs. The columns *"TIME"* and *"MEMORY"* present the running time and memory usage for comparing both initial and final AIGs using BDD and SAT approaches. The experiments were performed until the running time exceeds a limit of 27 hours. Notice that the running time of BDD method exceeds this limit in 8 of the 12 compared AIGs. The average running time of the BDD-based approach is 8.96 times slower than the SAT-based approach. Rather, SAT-based method was able to compare AIGs under the threshold time for most of the circuits. With respect to memory usage, notice that using BDDs needed 35% more memory than the SAT counterpart.

## 6. CONCLUSIONS

This paper presented a review for AIG equivalence checking. In this work, we analyzed both BDD and SAT approaches. Applying these two methods over a set of benchmark AIGs, the results show that the results show that almost 67% of the benchmark AIGs could not be checked using the BDD approach due to the running time exceeded the threshold of 27 hours. On the other hand, the SAT approach successfully verified 11 of the 12 benchmark AIGs (almost 67% of them in less the 1 second). The SAT-based approach also achieved a lower memory usage than the BDD-based approach. However, as equivalence checking is a NP-complete problem, it still does not exist an algorithm that efficiently checks the equivalence of AIGs for every input.

## 7. REFERENCES

[1] S. B. Akers. Binary Decision Diagrams. *IEEE Trans. on Computer*, 27(6):509–516, 1978.

[2] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. Release 20130425.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. of Design Automation Conference (DAC)*, 1991.

[4] R. K. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *Proc. of Int'l Symp. on Circuits and Systems (ISCAS)*, 1982.

[5] R. E. Bryant. Graph-based algorithms for Boolean functions manipulation. *IEEE Trans. on Computer*, 35(8):677–691, 1986.

[6] J. Cortadella. Timing-driven logic bi-decomposition. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 22(6):675–685, 2003.

[7] G. de Micheli. *Synthesis and optimization of digital circuits*. Tata McGraw-Hill Education, 2003.

[8] N. Een and N. Sörensson. MiniSat: A SAT solver with conflict-clause minimization. *Sat*, 5, 2005.

[9] S. H. Gerez. *Algorithms for VLSI Design Automation*. John Wiley & Sons Ltd, West Sussex, 1 edition, 1999.

[10] E. Goldberg, M. Prasad, and R. Brayton. Using sat for combinational equivalence checking. In *Proc. of Design, Aut. and Test in Europe (DATE)*, 2001.

[11] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Norwell, Massachusetts, 1 edition, 1996.

[12] Y. Hamadi, S. Jabbour, and L. Sais. Manysat: a parallel sat solver. *JSAT*, 6(4), 2009.

[13] C. Y. Lee. Binary Decision Programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[14] Y. S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient sat solver. In *Theory and Applications of Satisfiability Testing*, 2005.

[15] Y. Matsunaga. An efficient equivalence checker for combinational circuits. In *Proc. of Design Automation Conference (DAC)*, 1996.

[16] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In *Proc. of Design Automation Conf. (DAC)*, 2006.

[17] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive*