Restoring Circuit Structure from SAT Instances

Jarrod A. Roy
University of Michigan
EECS Department
Ann Arbor, MI 48109-2122
royj@eecs.umich.edu

Igor L. Markov
University of Michigan
EECS Department
Ann Arbor, MI 48109-2122
imarkov@eecs.umich.edu

Valeria Bertacco
University of Michigan
EECS Department
Ann Arbor, MI 48109-2122
vale@eecs.umich.edu

ABSTRACT

SAT solvers are now frequently used in formal verification, circuit test and other areas of EDA. In many such applications, SAT instances are derived from logic circuits. It is often assumed in the literature that circuit structure is lost when a conversion to CNF clauses is made [9]. We aim to examine this assumption. Specifically we formulate classes of combinational circuits that can be reproduced entirely from their SAT encodings. Using this knowledge, one may be able to apply Circuit-SAT techniques to a wider range of SAT instances and benefit from their improved performance on circuit-derived instances.

1. INTRODUCTION

Thanks to recent advances in algorithms for Boolean Satisfiability, SAT solvers are now routinely used in formal verification and circuit test, especially in equivalence checking, bounded model checking (BMC), automatic test pattern generation (ATPG), and microprocessor verification. Generic SAT-solvers made great strides over the last 8 years, with such breakthrough works as GRASP [11], Chaff [14] and BerkMin [7] that improved runtime by several orders of magnitude on various application-derived benchmarks. Such surprisingly good performance in solving an NP-complete problem spawned a series of new applications not only in EDA, but also in Artificial Intelligence and Operations Research, particularly in planning and scheduling. In a more recent development, primarily of relevance to circuit-based applications, generic SAT solvers were extended to explicitly use the circuit from which the input CNF instance was derived [9]. Again, runtime was improved by an order of magnitude on some classes of benchmarks, but with the caveat that circuit structure *must be known prior to solving*. These techniques include (i) fast parallel simulation on a small number of random inputs, (ii) detecting signal correlations by hashing, (iii) guiding the SAT solver to disprove the equivalence of correlated signals, and thus helping it to generate more concise and efficient conflict clauses. Notably, the authors assume that circuit structure is lost in the conversion to a SAT instance [9].

Our work addresses the issue of identifying logic circuit structure in a given SAT instance and offers a general algorithm to detect all occurrences of a logic gate in a given SAT instance. We prove that if a given SAT instance is derived from a combinational circuit of AND, OR, and NOT gates, then there is only one circuit that can lead to such a SAT instance. Furthermore, we contribute a specialized practical algorithm to reconstruct this circuit.

An EDA engineer using circuit-SAT solvers at work may wonder why one would want to first convert circuits to CNF formulas and then back to circuits. Indeed, when circuit information is readily available, this seems unnecessary and wasteful. However, as our experiments show, some CNF formulas may contain a large portion of circuit-derived clauses, and yet not be fully traceable to combinational circuits. An obvious class of examples is from property checking, where the circuit part describes the hardware, and the non-circuit part represents more general properties, such as "at most k out of n signals can be high". Additionally, our empirical data (e.g., on `Pret` benchmarks) show that circuit structure may emerge as a side-effect of mathematical constructs, unbeknownst to people performing the encoding. Another reason to bridge the gap between the CNF-SAT and circuit-SAT community is to facilitate the free flow of ideas, the exchange of solver implementations, and their evaluation in the context of different applications.

In the more general case where the SAT instance is not circuit-derived or involves un-oriented gates, such as XOR, our algorithms can identify the portions of the CNF formula that are compatible with a circuit structure. The recognition of such circuit sections may also facilitate the use of specialized SAT solvers on generic SAT problems, whether or not the reconstructed circuits are unique. In general, multiple candidate sub-circuits may share some clauses, making them incompatible. In this case, our algorithms extract only a non-overlapping set of sub-circuits.

Our general circuit-detection algorithm operates by translating the gate-matching problem into one of recognizing sub-hypergraph isomorphism. This translation is done by representing the SAT instance and the CNF-signature of a logic gate with directed hypergraphs. Further, directed hypergraphs are transformed to graphs with labeled vertices, facilitating the use of existing algorithms and software for the sub-graph isomorphism problem. Our contributions are backed up by solid empirical data, showing that our specialized algorithm is very scalable and can detect large amounts of circuit structure, even in SAT instances that are not circuit-derived.

The remainder of the paper is structured as follows. Section 2 addresses the necessary preliminaries. Section 3 describes a general algorithm for reconstructing logic gates from SAT instances. Next, Section 4 offers more efficient restoration of circuits with specific properties, and Section 5 extends these efficient techniques to a larger variety of circuits. Empirical results on circuit-detection in SAT benchmarks are described in Section 6. Section 7 wraps up the paper with conclusions and directions for further work.

2. PRELIMINARIES

Converting combinational circuits to CNF-SAT is straightforward [16, 12]. The resulting SAT instance will contain one variable for each primary input and one variable for the output of each gate (i.e., a variable corresponds to a wire in the circuit). The number of clauses in the resulting instance is determined by the number and type of gates in the circuit. For example, a NOT gate of the form $z = NOT(x)$ is transformed into the clauses $(x+z)(\overline{x}+\overline{z})$. An AND gate of the form $z = AND(x_1,\ldots,x_j)$, on the other hand, is trans-

formed into the clauses reported in the first row of the list below. The clauses for OR, NAND, and NOR gates are quite similar to that of AND gates [16, 12]:

- $z = AND(x_1,\ldots,x_j) \equiv \left[\prod_{i=1}^{j}(x_i+\overline{z})\right]\left(\sum_{i=1}^{j}\overline{x_i}+z\right)$
- $z = OR(x_1,\ldots,x_j) \equiv \left[\prod_{i=1}^{j}(\overline{x_i}+z)\right]\left(\sum_{i=1}^{j}x_i+\overline{z}\right)$
- $z = NAND(x_1,\ldots,x_j) \equiv \left[\prod_{i=1}^{j}(x_i+z)\right]\left(\sum_{i=1}^{j}\overline{x_i}+\overline{z}\right)$
- $z = NOR(x_1,\ldots,x_j) \equiv \left[\prod_{i=1}^{j}(\overline{x_i}+\overline{z})\right]\left(\sum_{i=1}^{j}x_i+z\right)$

The functionality of a logic gate can be described as follows.

Definition 1. The *characteristic function* of a logic gate with i inputs and o outputs is a Boolean function $\chi : B^{i+o} \rightarrow B$ such that, for each distinct assignment to all the inputs and output nodes, $\chi = 1$ iff the values at the output nodes are compatible with the gate functionality and the input assignments, otherwise $\chi = 0$.

As shown above, the conversion of a circuit to CNF consists of expressing and instantiating characteristic functions of individual gates in the CNF form. Note that a completely specified gate has a unique characteristic function, which may be represented by many possible CNF formulae. The formulae shown are minimal and most commonly used in deriving SAT instances from combinational logic circuits. In Section 4 we exploit this fact not only to detect the occurrences of such logic gates in a SAT instance, but also to identify which variables correspond to inputs and output of a gate. In the sequel we distinguish a special class of logic gates:

Definition 2. A gate is *unoriented* iff its characteristic function χ is symmetric, i.e., invariant under any permutation of its variables.

Examples of unoriented gates are XOR and XNOR, and any CNF formula obtained from them must be symmetric. Therefore, it is impossible to "guess" literals corresponding to inputs and outputs by only inspecting the CNF formula.

Example 1. Consider a 2-input XOR gate: $z = x \oplus y$. The characteristic function for this gate is $\chi_z = (z \oplus x \oplus y)$. The CNF sub-formula generated by the gate is: $(\overline{x}+y+z)(x+\overline{y}+z)(x+y+\overline{z})(\overline{x}+\overline{y}+\overline{z})$. Both expressions are symmetric and the variable corresponding to the output node is indistinguishable from the others. □

In the general case, the CNF representation of an n-input XOR gate $z = XOR(x_1,\ldots,x_n)$ contains the 2^n clauses of $n+1$ literals each where the number of negated literals is odd. This is because variables in an n-input XOR relation (where one of the variables is the result) must have even parity overall. Thus an assignment to the $n+1$ variables making up an XOR relation that has odd parity will not satisfy at least one of these clauses. Similarly, the CNF representation of an n-input XNOR gate $z = XNOR(x_1,\ldots,x_n)$ contains the 2^n clauses of $n+1$ literals each where the number of negated literals is even.

3. A CIRCUIT-DETECTION ALGORITHM

We now describe a general circuit-detector algorithm. It is based on gate matching for gates specified by a CNF formula as in Section 2, and identifies all occurrences of a gate in a SAT instance.

Definition 3. A *CNF-signature* of a logic gate is a CNF formula representing the characteristic function of the gate.

While a logic gate may have many distinct CNF-signatures, a given CNF-signature identifies no more than one gate.

The gate-matching algorithm is described below and used iteratively to reconstruct large circuits. As its first step, the circuit-detector identifies all occurrences of each gate. It then assembles such gate-matches into circuits. One approach is to connect compatible gate inputs and outputs, expanding the circuit as much has possible without gate overlaps. If a given SAT instance is fully derived from combinational logic, the initial circuit may be reconstructed, so that every variable is traced to a wire and every clause is a part of a gate's CNF-signature. However, in the more general case where at least some portions of the SAT instance are not circuit-derived, the result may consist of multiple, non-overlapping and non-connected logic networks surrounded by unmatched clauses. One may also have to choose among overlapping gate matches.

3.1 Matching a CNF-signature

To discover all occurrences of a given CNF-signature in a SAT instance, we reduce this problem to one of recognizing sub-graph isomorphisms.

Sub-graph isomorphism. Here one seeks a mapping of the vertices of a smaller graph *into* the vertices of a larger graph so as to map edges to edges. The sub-graph isomorphism problem is known to be NP-complete for general graphs [6]. However, practical algorithms are known with reasonable runtime and memory requirements in special cases, such as bounded-degree graphs [8, 10] and for typical inputs. The classic algorithm by Ullman [17] uses backtracking to dramatically prune the search space for irregular graphs. It is widely used and has been implemented in the VFLib graph matching library [19], along with other graph matching algorithms, such as one by Cordella *et al.* [4] with better performance and lower memory requirements.

CNF formulae and directed hypergraphs. Given the available theory and software for the sub-graph isomorphism problem, one would like to reduce the gate-matching problem to sub-graph isomorphism. We accomplish this by a series of transformations, from the original CNF form to directed hypergraph, to bipartite digraph and finally to undirected labeled graph, which we use as inputs to the sub-graph isomorphism algorithm. A hypergraph is a pair $H = (V,E)$, where V is a finite set of vertices and E is a set of hyperedges, $E_k = [v_1, v_2, \ldots v_k]$, that is, subsets of V of cardinality > 1. A directed hypergraph has hyperedges whose components have an associated direction. We represent the direction by prefixing each vertex in a hyperedge list with a $+$ or $-$ sign, depending is the direction is incoming to the vertex or outgoing, respectively. An example of a directed hypergraph is reported in Figure 3.1.a. We transform a CNF formula to a directed hypergraph by generating a vertex for each clause and a hyperedge for each variable. A hyperedge will connect all the vertices that share its corresponding variable. The direction of each component of the hyperedge will have a $+$ sign is the variable appears in the clause with positive polarity, it will have a $-$ sign, otherwise.

Example 2. Consider the following CNF formula: $(b+d+\overline{c})(c+a+\overline{b})(a+\overline{c})(d+\overline{a})$. Using the transformation just described, we construct a hypergraph with four vertices and four hyperedges as represented in Figure 3.1.a. In the Figure we labeled the hyperedges with the corresponding variable and the vertices with a number indicating the position of the corresponding clause. The only purpose of these labels is to improve readability and they are not part of the hypergraph. Observe the correspondence between the polarity of the literals and the direction marked on the hyperedges with arrows. □

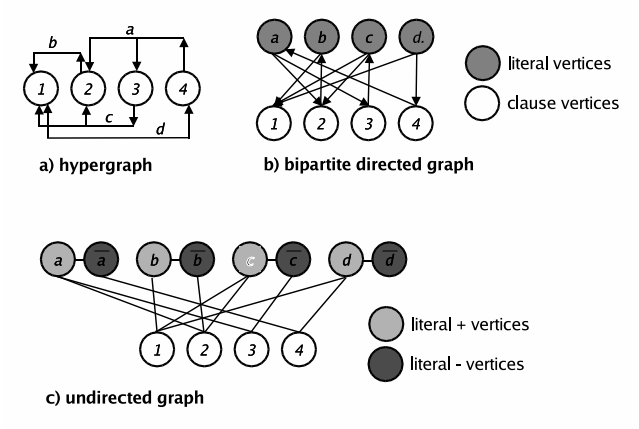

Figure 1: Conversion of the CNF formula $(b+d+\overline{c})(c+a+\overline{b})(a+\overline{c})(d+\overline{a})$ to multiple types of graphs.

So far, we can generate a canonical directed hypergraph from a CNF formula. Vice versa, given a directed hypergraph, it is straightforward to restore the corresponding CNF formula, once each hyperedge has been assigned to a variable. This one-to-one correspondence can be contrasted with a one-way translation of CNF formulas into undirected hypergraphs [1] by mapping clauses to hyperedges and literals to vertices, that ignores the polarity of literals.

Graph transformations. It is well known that an undirected hypergraph $H=(V_h,E_h)$ can be represented by a bipartite graph $BG=(V_v,V_e,E_b)$ if we map vertices to vertices, $V_h \rightarrow V_v$, and hyperedges to additional vertices, $E_h \rightarrow V_e$. A vertex in V_e is adjacent to a vertex in V_v iff the corresponding hyperedge is incident upon the corresponding hypergraph vertex. A graph such as *BG* is called the incidence graph of the hypergraph. Since in our specific context hypergraphs are directed, we can modify the construction by representing directed hypergraphs with directed bipartite graphs, where the direction of the edges connecting the vertices in V_v and V_e is the same as the direction of the corresponding hyperedge.

The final transformation step requires that we remove the directionality of the edges so that the resulting graphs can be inputs to the sub-graph isomorphism algorithm. A simple way to achieve this without loss of information is to construct a undirected graph $G=(V_v,V_{e+},V_{e-},E_g)$, where V_v are the same as in *BG*, and for each vertex in V_e there is one vertex in the V_{e+} set and one in the V_{e-} set. *G* has one undirected edge connecting each pair of (V_{e+},V_{e-}) vertices and one undirected edge for each directed edge of *BG*: where *BG* has a vertex from a V_v to a V_e, *G* has an edge from a V_v to a V_{e-}, if the edge in *BG* has opposite direction, *G* connects a vertex in V_v with one in V_{e+}. The total number of vertices in the final graph is $(\text{clauses}+2*\text{variables})$, while the total number of edges is $(\text{variables}+\text{literals})$.

Example 3. Figure 3.1.b reports the transformation of the hypergraph from the previous example to a directed bipartite graph. Figure 3.1.c shows that directed bipartite graph converted into an undirected graph. □

3.2 Restoring whole circuits

Earlier, we described gate detection as part of an overall algorithm to detect logic circuits in a given SAT instance. Now we turn to reconstructing the entire circuit from the detected gates.

The driving requirements when deciding if two logic gates sharing some literal should be connected are that 1) the inputs/outputs mappings of each gate should be compatible with the connection, and 2) no combinational loops should be generated as a result of the newly formed connection. For unoriented gates, any assignment compatible with the second requirement and with each gate having at least one input and output is acceptable and generates a feasible combinational circuit.

Example 4. Consider the following SAT instance: $(\overline{a}+b)(\overline{a}+c)(a+\overline{b}+\overline{c})(\overline{b}+a)(\overline{b}+c)(b+\overline{a}+\overline{c})$. The gate-matching algorithm can detect two AND gates: $a=bc$ and $b=ca$. However, the circuit-detector cannot connect the two gates at nodes *a* and *b* without generating a loop. The output of the algorithm in this case consists of two one-gate circuits. □

We wish to produce the largest possible circuit from a CNF instance such that there are no incompatibilities. One way to accomplish this objective is to map this problem to the maximal independent set (MIS) optimization problem. The MIS problem is an NP-complete problem for graphs that asks if there is a set of vertices of a given size in the graph such that none of the vertices are pairwise connected and such that adding one extra vertex to this set would break this property [6]. The MIS optimization problem is an extension of the MIS problem that asks for the largest MIS of a given graph.

To reduce our problem of finding the largest possible compatible circuit to the MIS optimization problem, we convert each detected gate to a vertex and place an edge between vertices if their corresponding gates are incompatible (share a CNF clause, for example). A solution to this MIS optimization problem gives us the largest circuit (in terms of number of gates) that is compatible with the SAT instance. The MIS optimization problem is NP-hard, but efficient heuristics exist. In the next section we enumerate a class of circuits and prove that they can be reconstructed quickly and unambiguously without the need of such an MIS technique.

4. CONVERTING AND-OR-NOT CIRCUITS TO CNF FORMULAS AND BACK

The general algorithm given in Section 3 above will correctly identify circuit structures from CNF instances, but it makes no guarantee of efficiency nor uniqueness of this identification. In this section we identify a class of circuits for which the circuit to CNF mapping given in Section 2 is a bijection and give a fast algorithm for these types of circuits.

We define an AND-OR-NOT circuit to be an acyclic combinational circuit composed of only AND gates, OR gates, and NOT gates. We do not impose any fanin or fanout count restrictions on these gates. We do impose the restriction that either the input or output of a NOT gate be an AND gate or an OR gate. This restriction does not allow circuits with long chains of NOT gates or disconnected NOT gates. While not being terribly restrictive, the reasoning for this restriction on NOT gates will become clear shortly.

Given an AND-OR-NOT circuit and a naming of the wires of the circuit, the method of encoding AND-NOT circuits as SAT instances defined in Section 2 will produce a unique CNF instance. We can easily see that this is true because the encoding offers no variability in how each individual piece of the circuit is encoded. We wish to be able to recover the initial circuit uniquely, so we prove the following:

THEOREM 1. *The mapping from AND-OR-NOT circuits with wire labellings to CNF instances (as defined in Section 2) is unique. In other words, if two circuits map to the same set of CNF clauses, the circuits must be identical.*

PROOF. Let us be given two circuits with wire labellings, Circuit 1 and Circuit 2 for convenience, such that their CNF mappings are the same. One first observation is that Circuit 1 and Circuit 2 must have the same number of wires (wires being either primary inputs or the outputs of gates). If, for example, Circuit 1 had more wires than Circuit 2, the CNF mapping of Circuit 1 would have more variables than the CNF mapping of Circuit 2. Since we have assumed the CNF mappings of Circuits 1 and 2 to be the same, this cannot be the case.

Let's examine each clause of the CNF mapping individually. We first begin with the 2-literal clauses. Each such clause may have zero, one, or two literals negated. If a clause has one negated literal, it came about because of an AND or an OR gate in the initial circuit. Let us say the clause is of the form $(\overline{a}+b)$. From the mapping, we can infer that this clause corresponds to either an AND gate with output wire a and an input wire b or an OR gate with output wire b and an input wire a. Both cannot be possible because this implies a cyclic initial circuit.

Otherwise (the clause has zero or two negated literals), the clause must have come about as a result of a NOT gate in the original circuit. In fact the mapping specifies that each NOT gate produces one 2-literal clause with zero negated literals and one 2-literal clause with two negated literals. Thus the number of NOT gates in the initial circuit is equal to the number of 2-literal clauses with zero negated literals which is also equal to the number of 2-literal clauses with two negated literals. Thus both Circuits 1 and 2 must have the same number of NOT gates. For each 2-literal clause with zero negated literals (or equivalently with two negated literals), let us remember a correspondence between the NOT gate in Circuit 1 that produced it and the NOT gate in Circuit 2 that produced it.

Secondly, we examine the rest of the clauses, i.e. the 3-or-more-literal clauses. According to the circuit to CNF mapping, the only way a clause with $n > 2$ literals can be produced is if the input circuit contains an AND gate with $n-1$ inputs or an OR gate with $n-1$ inputs. Further if this clause has exactly one positive (non-negated) literal, the gate in question must be an AND gate. Similarly, if this clause has exactly one negated literal, the original gate must have been an OR gate. There are no other possibilities. Thus Circuits 1 and 2 must have the same number of AND and OR gates. For each 3-or-more-literal clause, let us remember a correspondence between the gate in Circuit 1 that produced it and the gate in Circuit 2 that produced it.

This exhausts all the CNF clauses (since the mapping does not allow for any singleton clauses). As we have seen, each clause corresponds to exactly one gate in each of Circuits 1 and 2. We now know that Circuits 1 and 2 must have the same number of AND, OR, and NOT gates. Because they must have the same number of wires, they must also have the same number of primary inputs.

Now let us examine the connectivity of corresponding gates in Circuits 1 and 2. The output wire of each n-input AND and OR gate is uniquely determined by the $n+1$-literal clause that corresponds to it. In the case of the AND gate, the lone positive literal in the $n+1$-literal clause represents the wire which is the output of the gate. Likewise, the lone negated literal in the $n+1$-literal clause representing an n-input OR gate tells us which variable represents the wire that is the output of the gate. The n other variables in the $n+1$-literal clause are the inputs of their respective gates. Thus the connectivity of corresponding AND and OR gates in Circuits 1 and 2 are the same.

The connectivity in corresponding NOT gates in Circuits 1 and 2 is only slightly more difficult to ascertain. Because we have assumed Circuits 1 and 2 to be AND-OR-NOT gates as defined in Section 2, the input or output of each NOT gate must be connected to either an AND or an OR gate. Thus we can determine the input and output wires for each NOT gate by examining the other gate(s) to which it is connected. The 2-literal clause with zero negated literals corresponding to each NOT gate tells us which two wires are connected to the NOT gate. If one of these wires is also the output of an AND or OR gate, that wire is the input of the NOT gate and the other must be the output of the NOT gate. Otherwise, one of the wires must be an input of an AND or OR gate. That wire must be the output of the NOT gate and the other the input of the NOT gate.

Because the inputs and outputs of each AND and OR gate of Circuits 1 and 2 match, and the inputs and outputs of the NOT gates are based on these, the inputs and outputs of corresponding NOT gates in Circuits 1 and 2 must also match.

Circuits 1 and 2 have the same number of wire, AND gates, OR gates, NOT gates, and primary inputs. Also the connectivity of all corresponding gates match. Thus Circuits 1 and 2 must be the same. □

The proof of Theorem 1 suggests an algorithm for very efficiently reconstructing the unique AND-OR-NOT circuit from a CNF instance (assuming the CNF instance was derived from an AND-OR-NOT circuit in the first place). The algorithm proceeds exactly as the proof by examining each CNF clause individually. For 2-literal clauses, NOT gates are built only when the clause has zero negated literals (or equivalently two negated literals). For 3-or-more-literal clauses, one AND or OR gate is generated for each clause as appropriate. After all the gates are identified, the algorithm goes about determining the output wire of each gate. The non-trivial case comes when orienting NOT gates, but this has been already described in the proof. The last step, not mentioned by the proof, is to determine which of the wires are the primary inputs of the circuit. This is quite easy: any wire that is not the output of a gate is a primary input of the circuit.

Thus we see that AND-OR-NOT circuits can be reconstructed completely from their SAT encodings by a relatively simple-minded pattern matching algorithm that matches gates one at a time. Moreover, we can see that this simplistic algorithm runs in time linear in the size of the input SAT instance.

5. ALLOWING ADDITIONAL GATE TYPES

We now extend our fast, simple-minded pattern-matching algorithm for restoring circuit-structure by allowing for gates other than AND, OR, and NOT and point out several obstacles to such generalization posed by additional gate types. The results of this section are summarized in Table 1. Note that this is not an exhaustive listing of the gate types that can be detected. Other gates such as AOI gates used frequently in CMOS technologies can be detected but were not included due to space limitations.

5.1 NAND and NOR Gates

As we saw in Section 2, conversions from NAND and NOR gates to CNF clauses are well-defined. Thus it seems conspicuous that NAND and NOR gates were not part of the detection defined in Section 4.

In fact, allowing NAND and NOR gates is a little more tricky. All the literals in the $n+1$-literal clause of an n-input NAND gate are negated while all the literals in the $n+1$-literal clause of an n-input NOR gate are positive. This makes them fairly easy to recognize, but makes figuring out the orientation of the gate a little more difficult.

This is remedied by choosing one of the literals to be the output and checking this decision by testing the existence of the proper 2-literal clauses. For example assume that the clause $(x+y+z)$ is

found in the instance. We recognize this as a possible NOR gate, and so try to find the output of the gate. We choose x first and test for the existence of the clauses $(\overline{x}+\overline{y})$ and $(\overline{x}+\overline{z})$. If both these clauses exist, x is the output of the gate. Otherwise x cannot be the output of the gate and try y, etc. until we discover the output of the gate. If we assume a good hash function, testing for the existence of the 2-literal clauses can be done in constant time which makes testing for NAND and NOR gates take time proportional to the number of inputs of the gate.

Thus we see that NAND and NOR gates can be properly detected, but slightly less efficiently than AND and OR gates. Pseudocode for detecting these types of gates is given in Figure 2.

```
1   foreach clause c with > 2 literals
2       foreach literal l in c
3           foreach literal m in c, m ≠ l
4               if (l̄ + m̄) not found then next l
5           // found a gate, l is its output
6           // the inputs correspond to the other
7           // literals of c; now to determine its type
8           if l negated
9               if remaining literals are negated
10                  // gate is NAND
11              else // gate is OR
12          else // l not negated
13              if remaining literals are negated
14                  // gate is AND
15              else // gate is NOR
16          record this gate's inputs and outputs
```

Figure 2: Detecting AND, OR, NAND, and NOR gates.

5.2 XOR and XNOR Gates

XOR and XNOR gates are common in circuit designs. They can be constructed using only AND, OR, and NOT gates, but at a significant size penalty. For example one 2-input XOR gate can be equivalently represented with two 2-input AND gates, one 2-input OR gate, and 2 NOT gates. The SAT encoding of one XOR gate requires three variables and four 3-literal clauses while the equivalent AND-OR-NOT encoding needs seven variables, three 3-literal clauses, and ten 2-literal clauses.

Now we consider the possibility of allowing XOR and XNOR gates to be contained within the circuit to be encoded. Obviously if we can allow XOR and XNOR gates, there is good potential for savings in terms of circuit and SAT instance size. First we must make sure that the clauses of XOR and XNOR gates can be recognized from a SAT instance. Both n-input XOR and XNOR gates are converted into 2^n CNF clauses of $n+1$ literals each, as we saw in Section 2.

The literals in each of the clauses of an n-input XOR gate form a very interesting pattern. The 2^n clauses of $n+1$ literals representing an n-input XOR gate are the ones with an odd number of negated literals. Similarly, the 2^n clauses of $n+1$ literals with an even number of negated literals form the CNF representation of an n-input XNOR gate.

Thus, when examining a clause with $n+1$ literals, it is easy to detect if it may be part of an n-input XOR or XNOR gate (since it cannot possibly be part of both). After making such a determination, all one must do is check for the other necessary clauses (i.e. the other clauses with an odd number of negated literals for an n-input XOR gate and the clauses with an even number of negated literals for an n-input XNOR gate). If all of the other necessary clauses are found, then we are guaranteed that an n-input XOR or XNOR gate has been detected. The amount of time required to detect an XOR or XNOR gate seems like it may quite large since an n-input gate may possibly be detected 2^n times. All we need to do is mark each clause if we search for it and find it. When we get to a clause that has already been marked, we can be sure that if it is part of an XOR/XNOR gate, that XOR/XNOR gate has already been detected. Generating the other necessary clauses will take time linear in the size of the clause for each other necessary clause, so detecting XOR gates should take time similar to detecting NAND and NOR gates.

Once XOR and XNOR gates are recognized, we must calculate the orientation of the gate. *However, despite what their popular symbols may suggest, XOR and XNOR gates do not have inherent orientations.* Therefore, SAT encodings of such gates do not imply which variable is the output of the gate, much like the encoding of a NOT gate. To get around this difficulty for NOT gates in Section 4, we required that either the input or output of a NOT gate be connected to an AND or OR gate. To allow XOR and XNOR gates, we need to make a similar, but slightly more general, restriction.

To clearly explain the restriction, let us divide up gates into two disjoint groups: those whose orientation can be known from their SAT encodings and those whose orientation can only be derived from their surrounding gates. As we have seen so far, AND, OR, NAND, and NOR gates fall into the former group and NOT gates as well as XOR/XNOR gates fall into the latter group. The restriction that we make is that if a gate of a circuit falls into the latter group, one must be able to determine the orientation of all its fanins or all of its fanouts. To make this more clear, we must be able to impose an ordering on all the unoriented gates such that by proceeding in this ordering we can determine the orientation of the first gate in the ordering from the orientations of its adjoining gates, and then the second gate, etc. until all gates in the circuit are properly oriented.

To see why this restriction is sufficient, let us consider the task of determining the orientation of XOR/XNOR gates that have been guaranteed to have this property. Say that we are trying to determine the orientation of a detected 2-input XOR gate which has wires x, y, and z. First let us assume that the orientation of the fanins of this gate can be determined and further have been. This guarantees that exactly two of the wires have been marked as outputs of other gates. Thus the third wire of this gate must be its output and we have determined the orientation of this gate. Now let us assume that the orientation of the fanouts of this gate can be determined and have been. This guarantees that exactly one of the wires has been marked as an input to all other gates to which it connects. If a wire is connected to this gate and several others and all the other gates claim it as an input, this wire must be the output of this gate. Thus we have determined the orientation of this gate.

The ordering in which we attempt to orient XOR/XNOR gates will make a difference in how quickly we determine the orientation of all the gates, but our assumptions guarantee that at least one ordering does exist so we will be able to eventually order all gates. A naive process of trying to orient gates by using a static ordering can take a number of steps quadratic in the number of unoriented gates in the worst case, but a smarter breadth first technique can orient in linear time.

Note that this restriction on XOR/XNOR gates and method to figure out their orientations is not necessarily the only way nor the best way to do it. It may even be possible that its not necessary to find the orientations of XOR/XNOR gates for them to be useful. Different ways of using XOR/XNOR gates is one direction of further research.

Thus we can conclude that circuits with properly constrained NOT and XOR/XNOR gates can be reconstructed from their SAT encodings. Pseudocode for detecting these XOR and XNOR gates is given in Figure 3.

```
1   foreach clause c with > 2 literals
2       if c marked then next c
3       for i = 2; i ≤ size(c); i = i+2
4           foreach set s of i literals in c
5               clause c' = ∅
6               foreach literal l in c
7                   if l ∈ s then add l̄ to c'
8                   else add l to c'
9               if c' in instance then mark c'
10              else next c
11      if c has odd number of negated literals
12      then // gate is XOR
13      else // gate is XNOR
14      record this gate's information
```

Figure 3: Detecting XOR and XNOR gates.

5.3 Majority of 3 Gates

The majority of 3 gate (MAJ3) is a gate type that has many interesting theoretical implications. Given three boolean inputs, the MAJ3 gate outputs the most common of its inputs. It can be defined as $z = MAJ3(x_1, x_2, x_3) \equiv z = x_1x_2 + x_1x_3 + x_2x_3$. Given this definition, the SAT encoding is found to be six clauses of three literals each: $(\overline{x_1} + \overline{x_2} + z)(\overline{x_1} + \overline{x_3} + z)(\overline{x_2} + \overline{x_3} + z)(x_1 + x_2 + \overline{z})(x_1 + x_3 + \overline{z})(x_2 + x_3 + \overline{z})$.

The SAT encoding of MAJ3 gates is much different than the other gate types we have already examined. One large difference, which makes this type of gate more difficult to detect, is the fact that none of the clauses contain all of the variables associated with the gate. All of the previous n-input gates had at least one clause with $n+1$ literals. In this case, we will need to examine pairs of clauses of the proper size to figure out all of the variables associated with the gate.

Given that we will have to examine pairs of 3-literal clauses, this process will require more effort since it could run in quadratic time in the worst case. To minimize the number of pairs to consider, we can make use of another property of the SAT encoding of the MAJ3 gate. If a clause has any chance of being part of a MAJ3 gate encoding, it must have either exactly one or two negated literals and a similar clause with all of its literal negations reversed must be present. In other words, if $(\overline{x_1} + \overline{x_2} + z)$ has a chance of being part of a MAJ3 gate encoding, $(x_1 + x_2 + \overline{z})$ must also be a part of the SAT instance. Thus we can pass over all the 3-literal clauses and record only those which have a possibility of being in a MAJ3 encoding. In practice this eliminates a significant portion of the 3-literal clauses and the number of pairs to check is small.

Once we have only those clauses which are possible, we consider them pairwise. First we can check to see if the two clauses match for their output (the output is always distinguished in each of the clauses as being the sole negated or sole positive literal), we may proceed. Next if we can determine exactly three inputs from these two clauses (from the four remaining literals), we can generate and check for the other required clauses. If all required clauses are found, we are guaranteed to have detected a MAJ3 gate. Determining orientation is automatic since the output is always distinguished in each of the clauses. Thus it is possible to detect MAJ3 gates, but it takes more effort than other gate types.

Gate type	Difficulty of restoring circuit structure
OR *and* AND	Straightforward pattern-matching
NOR *and* NAND	Pattern-matching with back-tracking
NOT, XOR *and* XNOR	Can be detected by straightforward pattern-matching, but w/o orientation, which can only be determined in the context of other gate types
MAJ3	More advanced pattern matching with back-tracking

Table 1: The relative difficulty of detecting particular types of logic gates in CNF-SAT formulas. Note that this is not an exhaustive listing of detectable gates.

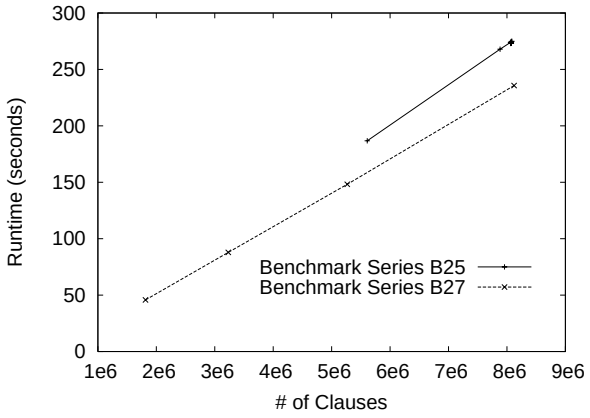

Figure 4: Runtime vs. SAT instance size on Velev's B25 and B27 series [18] of benchmarks.

6. EMPIRICAL STUDIES ON RESTORING CIRCUIT STRUCTURE

The techniques presented above can be generalized to detect if an arbitrary SAT instance contains circuit structure and restore such structure. Our initial belief was that many common SAT benchmarks contain underlying circuit structure that may or may not have been known when initially constructing the instances. Therefore, we implemented the gate-matching algorithms and empirically evaluated their performance on the well-known DIMACS benchmarks [5] and the `XOR-Chain` series from the SAT 2002 competition [15]. The rationale behind this experiment is that successful detection of circuit structure would facilitate the application of specialized Circuit-SAT solvers [9] and potentially speed up SAT-solving.

Table 2 summarizes the results of our experimentation and lists, one per line, cumulative data for seven benchmark series. As is evident from the table, benchmarks in these series exhibit significant circuit structure. It is interesting to note that while the `Bf` and `Ssa` series are circuit-derived (and relate to testing for bridging faults and single-stuck-at faults), benchmarks in the `Hanoi` series describe a famous logic puzzle commonly studied in Artificial Intelligence (the Towers of Hanoi problem). The large number of XOR gates in `Parity` benchmarks is no surprise, but we did not expect to find an even larger proportion of XORs in `Dubois` benchmarks, whose official description reads: "Randomly generated SAT instances". `Pret` benchmarks also exhibit a large proportion of XOR gates, and thus their official description "Encoded 2-colouring forced to be unsatisfiable" offers additional food for

Benchmark series	# of benchmarks	# of variables	# of clauses	% variables in simple gates	% clauses in simple gates	% variables in XOR/XNORs	% clauses in XOR/XNORs	Detection runtime
Bf	4	5793	16566	54.29%	22.12%	1.18%	0.54%	0.43
Dubois	13	1275	3400	0%	0%	100%	100%	0.09
Hanoi	2	2041	12272	43.22%	10.19%	0%	0%	0.37
Parity	30	24267	83330	33.17%	13.58%	88.35%	68.42%	2.68
Pret	8	840	2240	0%	0%	100%	100%	0.09
Ssa	8	7828	17669	47.25%	18.57%	1.45%	0.69%	1.09
XOR-Chain	27	4554	12126	0%	0%	100%	99.55%	0.38

Table 2: Circuit structure detected in selected DIMACS [5] and SAT2002 [15] benchmarks. Runtime for the structure detection is given in seconds.

Benchmark series	% variables remaining	% clauses remaining	% variables in simple gates	% clauses in simple gates	% variables in XOR/XNORs	% clauses in XOR/XNORs	Hypre runtime	Detection runtime
Bf	22.31%	31.17%	75.31%	46.41%	0.33%	0.12%	0.49	0.23
Dubois	100%	100%	0%	0%	100%	100%	0.05	0.09
Hanoi	64.97%	66.35%	52.24%	12.56%	0%	0%	0.23	0.24
Parity	54.99%	67.01%	30.70%	17.09%	100%	83.24%	1.47	2.13
Pret	100%	100%	0%	0%	100%	100%	0.03	0.09
Ssa	8.91%	8.33%	58.21%	29.86%	9.25%	4.74%	0.34	0.17
XOR-Chain	99.41%	99.55%	0%	0%	100%	100%	0.17	0.38

Table 3: Circuit structure detected in selcted DIMACS [5] and SAT2002 [15] benchmarks after applying the Hypre SAT preprocessor [2]. Runtimes for the preprocessor and the structure detection are given in seconds.

thought. Indeed, each edge in a graph that is to be 2-colored implies a mutual exclusion clause that may end up being a part of an XOR gate. Although XOR gate presence in the `XOR-Chain` benchmark series is also not surprising, it is worth note that the XOR-Chain series of benchmarks contains the benchmark that won an award at SAT 2002 for the smallest unsolved benchmark [15] and circuit structure detection takes very little time on them.

It is common practice to apply specialized SAT preprocessors on CNF-SAT instances in order to improve SAT-solving performance. One such preprocessor is Hypre [2]. Hypre applies the HypBinRes inference rule as well as equality reduction to SAT instances. HypBinRes is used to create new binary clauses which are further used to detect if literals are forced to be equivalent to other literals. Equality reduction is used to simplify the given instance based upon the detected equivalences. While Hypre does not guarantee optimal simplification, it can be more effective than exact logic simplifiers, such as Espresso-exact [13], since it can remove variables from the instance and is fairly scalable in practice.

We believe that circuit-to-CNF encodings are minimal (while the circuits themselves may not be minimal), so we were interested in seeing the effects of preprocessing on instances with circuit structure. We ran the Hypre preprocessor on the instances from Table 2 and the results are summarized in Table 3. In all but the `Dubois` and `Pret` series, which are fully comprised of XOR and XNOR gates, Hypre was quite effective in reducing the size of the SAT instances. In practically all cases, the percentage of circuit structure in the instances increased after the preprocessing simplification. This implies that the circuit structure in the instances was more difficult to simplify than the non-circuit part.

During our testing of the DIMACs benchmarks, we noticed that extracting circuit structure took relatively little time. To further stress-test our gate-matching algorithms and see how well they scale with the size of CNF formulas, we ran them on Velev's B25 and B27 series of benchmarks derived from formal verification of VLIW processors [18] on a 1.2GHz machine with 2.0 GB of RAM. The benchmarks ranged from 1.8 million to 8.1 million clauses and contained significant circuit structure (more than 94% of variables were part of circuit structure). A graph of runtime versus the number of clauses in the benchmark is shown in Figure 6. The figure shows the runtime to be roughly linear, so our asymptotic calculations of runtime match the empirical results.

Lastly, we decided to compare the runtime of circuit structure extraction with that of SAT solving. We compared runtimes of ZChaff [14] and two modes of the Circuit-SAT solver from [9], using the runtimes listed in [9]. Our runtimes were generated on a machine comparable to the one used in [9]. The results are summarized in Table 4. We can easily see that the runtime for circuit structure extraction is much smaller than the runtime of ZChaff. The Circuit-SAT solver was run in two modes: one with implicit learning and the other with explicit learning. The total runtime for implicit learning mode is the sum of the Implicit and Simulation columns. Similarly, the total runtime for the explicit learning mode is the sum of the Explicit and Simulation columns. Thus we see the runtime for circuit structure extraction is also small when compared to both modes of the Circuit-SAT solver. Thus using circuit structure extraction is certainly viable for speeding up SAT solving.

7. CONCLUSIONS AND FURTHER WORK

The objective of this paper was to shed the light on the misconception that all circuit structure is lost when converting from combinational circuits to SAT instances. It may be true that some structure can be lost depending upon the encoding (for example in the case of chains of XOR and XNOR gates), but we have shown that it's not always the case and summarized the relative difficulty of detecting particular gate types in Table 1. We have articulated that an arbitrary combinational circuit can be encoded as a CNF-SAT instance so that its circuit structure is preserved and can readily be extracted. In particular, we have described algorithms for restoring circuit structure from CNF formulas and empirically shown its success and scalability on very large benchmarks. *A major con-*

Circuit	ZChaff	Implicit	Explicit	Simulation	Extraction
9Vliw001	1057	567	793	93	10
9Vliw004	953	804	1011	92	10
9Vliw005	3126	740	1314	88	10
9Vliw007	140	286	855	110	11
9Vliw008	1450	239	1914	114	12
9Vliw009	1006	784	829	117	10
9Vliw010	867	329	1897	96	10
9Vliw015	2209	985	1270	97	10
9Vliw017	1007	175	913	109	10
9Vliw019	2936	849	1448	129	10
9Vliw021	1666	1069	1345	99	10
9Vliw024	1375	965	1282	107	9

Table 4: Comparing the runtime of circuit structure extraction to SAT solving by ZChaff [14] and the Circuit-SAT solver from [9]. The Circuit-SAT solver was run in two modes: one with implicit learning and the other with explicit learning. The total runtime for implicit learning mode is the sum of the Implicit and Simulation columns. Similarly, the total runtime for the explicit learning mode is the sum of the Explicit and Simulation columns. All runtimes are in seconds. Runtimes except for structure extraction are taken from [9].

clusion of our work is that CNF-based SAT solvers should not be considered a priori *inferior to Circuit-based SAT solvers because they have access to the same information.* Even more importantly, we have shown that some popular CNF-SAT benchmarks contain a large amount of circuit structure, while apparently not being entirely circuit-derived. This justifies extending the more general CNF-SAT solvers to account for circuit structure, rather than fall back on pure-circuit-SAT solvers and demand that the original circuit information be provided.

Our on-going and future work can be summarized as follows

- As mentioned in Section 5.2, there are different potentially better ways of handling XOR and XNOR gates. Our current method makes a very specific restriction on where XORs can be located in the circuit and how they can be connected. We believe there are other situations where XORs can be used even though their orientations may not be able to be unambiguously determined. We have also left it undetermined as to what should be done when XOR gates are present in the circuit but they do not conform to the restriction. One method is to remove the offending XOR gates and replace them with simpler gates, if the entire structure of the input circuit is absolutely necessary.

- Another approach to disambiguating the orientation of CNF-encoded XOR gates is to look at the algorithms using this orientation. For example, in [9], circuit information is used in random simulation that allows one to detect correlations between different wires/variables. Since such correlations do not inherently carry directionality information, they can be produced with any valid input/output assignment of an ambiguous circuit. However, it remains to be seen if such various I/O assignments perform equally well with the circuit-SAT algorithm from [9].

8. REFERENCES

[1] F. A. Aloul, I. L. Markov, and K. A. Sakallah, "Faster SAT and Smaller BDDs via Common Function Structure," *Proc. Intl. Conf. Computer-Aided Design*, pp. 443-448, 2001.

[2] F. Bacchus and J. Winter, "Effective Preprocessing with Hyper-Resolution and Equality Reduction", *Sixth Intl. Symp. on Theory and Applications of Satisfiability Testing 2003*.

[3] A. Biere and W. Kunz, "SAT and ATPG: Boolean Engines for Formal Hardware Verification", *Proc. Intl. Conf. on Computer-Aided Design 2002*.

[4] L. P. Cordella, P. Foggia, C. Sansone and M. Vento, "Evaluating Performance of the VF Graph Matching Algorithm", *Proc. of Intl. Conf. on Image Analysis and Processing*, 1999.

[5] DIMACS Benchmark Set for SAT, `http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html`

[6] M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-completeness", Freeman & co., New York, 1979.

[7] E. Goldberg and Y. Novikov, "BerkMin: A Fast and Robust SAT Solver", *Proc. Design Autom. and Test in Europe (DATE) 2002*, pp. 142-149.

[8] J. E. Hopcroft and J. K. Wong, "Linear Time Algorithm for Isomorphism of Planar Graphs", *Proc. Annual ACM Symp. Theory of Computing*, 1974.

[9] F. Lu et al., "A Circuit SAT Solver With Signal Correlation Guided Learning", *Proc. Design, Autom. and Test in Europe 2003*.

[10] E. Luks, "Isomorphism of Graphs of Bounded Valence Can Be Tested in Polynomial Time", *Journal of Computer System Science*, 1982.

[11] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A New Search Algorithm for Satisfiability", *IEEE Trans. On Computers*, vol. 48, no. 5, May 1999, pp. 506-521.

[12] J. P. Marques-Silva and K. A. Sakallah, "Boolean Satisfiability in Electronic Design Automation," In *Proc. Design Autom. Conf.*, June 2000, p. 675-680.

[13] P. C. McGeer, J. V. Sanghavi, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "ESPRESSO-Signature: A New Exact Minimizer for Logic Functions," *IEEE Trans. on VLSI*, 1(4), December 1993, pp. 432-440.

[14] M. W. Moskewicz, et al., "Chaff: Engineering an Efficient SAT Solver", *Proc. Design Autom. Conf.*, 2001, pp. 530-535.

[15] L. Simon, D. LeBerre, and E. Hirsch, "The SAT2002 Competition," `http://www.satlive.org/SATCompetition/2002/onlinereport/`

[16] G. S. Tseitin, "On the Complexity of Derivation in Propositional Calculus," in *Studies in Constructive Math. and Mathematical Logic*, Part 2, Consultants Bureau, New York, London, 1968, pp. 115-125.

[17] J. R. Ullman, "An Algorithm for Subgraph Isomorphism," *Journal of the ACM*, vol. 23, pp. 31-42, Jan. 1976.

[18] M. Velev, "Microprocessor Verification SAT Benchmarks", `http://www.ece.cmu.edu/~mvelev/sat_benchmarks.html`

[19] The VFLib Graph Matching Library, `http://amalfi.dis.unina.it/graph/db/vflib-2.0/doc/vflib.html`