Efficient Circuit to CNF Conversion

Panagiotis Manolios and Daron Vroon

College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332, USA
`http://www.cc.gatech.edu/home/{manolios,vroon}`

Abstract. Modern SAT solvers are proficient at solving Boolean satisfiability problems in Conjunctive Normal Form (CNF). However, these problems mostly arise from general Boolean circuits that are then translated to CNF. We outline a simple and expressive data structure for describing arbitrary circuits, as well as an algorithm for converting circuits to CNF. Our experimental results over a large benchmark suite show that the CNF problems we generate are consistently smaller and more quickly solved by modern SAT solvers than the CNF problems generated by current CNF generation methods.

1 Introduction

The recent drastic improvements to SAT solving technology have led to its wide applicability in domains ranging from hardware and software verification to computational biology to AI planning. While the actively developed SAT solvers overwhelmingly require input to be in CNF, most of the applications that use SAT technology have problems that are more naturally expressed as Boolean combinational circuits. In order to use current SAT solvers, users are therefore required to generate CNF and they tend to do this using variants of the Tseitin algorithm, *e.g.*, this is the case with Barcelogic Tools [8] and Yices [2].

In this paper we introduce NICE dags, a new data structure for representing circuits. We also introduce a new algorithm for translating from NICE dags to CNF. We have compared our algorithm to both the Tseitin algorithm and to Jackson and Sheridan's state-of-the-art algorithm, using a benchmark suite containing over 8,000 problems from various domains. Our extensive evaluations show that the translation from circuits to CNF can significantly impact overall SAT solving time and that our algorithm leads to significant time savings over the other two approaches. In fact, for numerous problems that our approach can easily handle, both algorithms generate CNF on which SAT solvers time out.

The work reported in this paper is implemented in the publicly available Bit-level Analysis Tool (BAT) [7]. BAT provides a high-level, feature-rich, type-safe language that includes user-defined functions, arbitrary sized bit vectors and operations on them, existential arrays, etc. With the results reported in this paper, we are able to provide potential users of SAT technology a much higher-level interface than current CNF-based SAT solvers. This makes it much easier to experiment with, use, and deploy SAT technology, while still being able to take advantage of improvements to the actively developed SAT solvers, which are mostly CNF-based.

J. Marques-Silva and K.A. Sakallah (Eds.): SAT 2007, LNCS 4501, pp. 4–9, 2007.

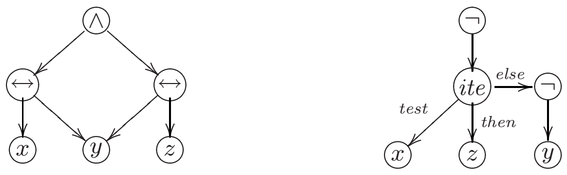


Fig. 1. Two examples of NICE dags

2 Related Work

Most modern CNF conversion algorithms are variations of the one originally discovered by Tseitin [9]. This algorithm converts dags composed of internal vertices labeled with $\wedge$, $\vee$, and $\neg$, as well as external vertices labeled with variable names. The algorithm introduces a new variable, x, for each internal node, v that is a child of a $\vee$ node, and adds the constraint $x \leftrightarrow v$ to the original SAT problem. This is a linear algorithm.

The Jackson-Sheridan algorithm for conversion to CNF was introduced in 2004 [5]. It takes a Reduced Boolean Circuit (RBC) as input, and is built around a heuristic that is used to decide when to introduce variables for internal vertices. The aim is to minimize the number of clauses in the resulting CNF. However, $\leftrightarrow$ vertices are rewritten into conjunctions of disjunctions before conversion into CNF, which results in the loss of information that can lead to the generation of fewer intermediate variables. The algorithm is quadratic.

Brummayer and Biere have recently introduced an algorithm for translating from AIGs (And-Inverter Graphs) to CNF [1]. The techniques used are mostly orthogonal to ours, and, for future work, it would be interesting to explore the incorporation of some of their ideas into our algorithm.

Another CNF translation that deals directly with *ite* vertices is presented by Velev in [10]. Our algorithm subsumes this algorithm, *e.g.*, we merge nested *ite* vertices in all the ways that Velev's algorithm does and more. Also, unlike Velev's algorithm, we constrain intermediate variables with implications rather than equivalences, which significantly reduces the number of clauses created when a new variable is introduced.

3 NICE Dags

The circuit representation that our CNF translation algorithm takes as input is the *Negation, Ite, Conjuction, and Equivalence dag (NICE dag)*, which contains external variable nodes, and as the name suggests, internal nodes labeled with $\neg$, $\leftrightarrow$, $\wedge$, and *ite*. The three outgoing edges of an *ite* are labeled with *test*, *then*, or *else*, as appropriate. As with RBCs, NICE dags are further constrained to maximize sharing. For example, no two nodes in the dag are allowed to have the same label and children, and no *ite* node can have a *test* or *then* child labeled with $\neg$. Two example NICE dags are given in Figure 1.

```
cnf ($(V, E)$)
  pseudo-expand ($(V, E)$)
  count-shares ($(V, E)$)
  $CL := \emptyset$
  for all $v \in V$ do
    clauses+ ($v$) := $null$
    clauses- ($v$) := $null$
  $C$ := cnf+ ($source((V, E))$)
  return $C \cup CL$
```

Fig. 2. Main CNF conversion function

4 CNF Conversion

In order to understand our CNF conversion algorithm, it is important to understand the following terminology. The *number of shares* of a node is the number of incoming edges the node has. A *path* is a sequence of vertices, $v_1, v_2, \ldots, v_n$ such that v_1 is the root of the dag and each consecutive pair of vertices forms an edge. The path polarity of a given path is *negative* if it contains an odd number of $\neg$ nodes, and *positive* otherwise. The *number of negative (positive) shares* for a node is the number of predecessors of the node that are the last vertex in a path of negative (positive) polarity. Note that a predecessor vertex can appear in a path of negative polarity and a path of positive polarity, so the the number of negative and positive shares can add up to more than the total number of shares.

Our central CNF conversion algorithm is given in Figure 2. It takes a NICE dag as input and returns a set of clauses that are equisatisfiable to the input. The algorithm begins with the *pseudo-expansion* of the the *ite* and $\leftrightarrow$ nodes of the NICE dag. We will return to this function momentarily.

The next function, `count-shares`, marks each node with its number of negative and positive shares. This is followed by initialization. A global variable, CL, is initialized to $\emptyset$. CL will contain the clauses constraining the variables introduced for internal vertices. All of the vertices, $v \in V$ are marked with a `clauses+` and `clauses-` value of *null*. These will eventually contain the clauses generated for v and $\neg v$, respectively.

The core of the algorithm, which is too long to give here, consists of two functions, `cnf+` and `cnf-`, which take a vertex, v, and compute `clauses+(`v`)` and `clauses-(`v`)` respectively. These functions recursively visit the children of v to compute their clause representations, and then combine the resulting clause sets in the appropriate way. For example, when applied to a $\wedge$ node, u, `cnf+` calls `cnf+` on all the children of u and then unions their clause lists together to form the clause list for u. A $\neg$ node, w, is processed by `cnf+` (`cnf-`) by applying `cnf-` (`cnf+`) to its child. Another way to see this is that `cnf+` and `cnf-` process the NICE dag using depth-first search, keeping track of the polarity of the path that led to the current node, and post-processing the node accordingly. The `cnf+` and `cnf-` algorithms have the following key features:

- The decision to introduce variables is made separately for a vertex and its negation, and the variable is constrained with an implication rather than an equivalence if it only represents a vertex or its negation, and not both. For example, if a variable, x is introduced for v, then the constraint $x \rightarrow v$ is added to CL. If we later decide that a variable is needed for $\neg v$, we add the constraint $\neg x \rightarrow \neg v$.
- When forming disjunctions, we use a similar heuristic to Jackson-Sheridan for deciding when to introduce new variables [5].
- A variable is always introduced for v ($\neg v$) if the number of positive (negative) shares for v is more than 1 and $|\texttt{clauses+}(v)| > 1$ ($|\texttt{clauses-}(v)| > 1$).
- *ite* and $\leftrightarrow$ are interpreted as conjunctions of disjunctions regardless of their polarity. For example, `cnf+(if` u `then` v `else` w`)` is computed by computing `cnf+(`$(\neg u \vee v) \wedge (u \vee w)$`)` and `cnf-(if` u `then` v `else` w`)` is computed by computing `cnf+(`$(\neg u \vee \neg v) \wedge (u \vee \neg w)$`)`. This leads to a smaller CNF translation.

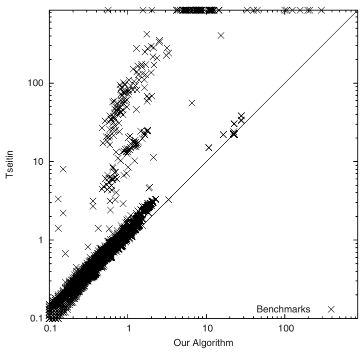
Note that this last case introduces some extra complexity to the algorithm. On the one hand, we want to translate *ite* and $\leftrightarrow$ nodes, as well as their negations, as conjunctions of disjunctions. This means that the translations of these nodes and their negations are no longer syntactic negations of each other. That is, the negation of an *ite* node, does not simply translate to a $\neg$ node whose child is the positive translation of the *ite*, since this would be a disjunction of conjunctions. That is why we do not expand *ite* and $\leftrightarrow$ nodes before the translation. This way, we can keep track of the fact that these translations are negations of each other by maintaining `clauses+` and `clauses-` for the original *ite* or $\leftrightarrow$ node.

On the other hand, in the example above, what if $u \vee w$ appears elsewhere in the dag? In this case, we will lose some sharing information if we do not expand the *ite* node before CNF conversion, since we will have another instance of $u \vee w$ when we do expand it. This is why we have the `pseudo-expand` function that starts our CNF translation algorithm. This function will mark the example above with new “pseudo-arguments” called `args+` and `args-`. The `args+` mark is set to the pair of vertices representing $(\neg u \vee v), (u \vee w)$. The `args-` mark is set to the pair of vertices representing $(\neg u \vee \neg v), (u \vee \neg w)$. Then, when counting shares, we count the shares of the “pseudo-arguments” of *ite* and $\leftrightarrow$ formulas rather than the actual arguments. This allows us to capture the appropriate sharing information without losing the negation information for *ite* and $\leftrightarrow$ and nodes.

In general, `cnf` $((V, E))$ has a running time of $O(|V|^2)$. However, as we will see in our experimental results, our algorithm runs faster than Tseitin's algorithm, which is linear, showing that our algorithm is linear is practice, at least for the benchmarks tested.

5 Experimental Evaluation

We implemented Tseitin's algorithm [9], Jackson and Sheridan's algorithm [5], and our algorithm in the Bit-level Analysis Tool (BAT). BAT is a tool for solving



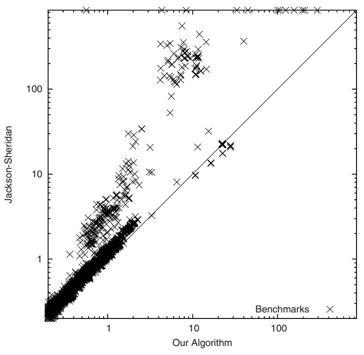


Fig. 3. Scatter plots comparing our algorithm to Tseitin and Jackson-Sheridan

formulas over the theory of bit vectors and existential arrays [7,6]. After processing arrays, BAT converts the resulting circuit into a NICE dag, which it uses to perform further simplifications. Finally, it converts the NICE dag to CNF.

We ran all three algorithms over a suite composed of 8,284 benchmarks. These include refinement theorems for two, three, and five stage pipeline machines, a correctness theorem for the Instruction Cache RAM unit from the Sun PicoJava II microprocessor, a theorem involving the out-of-order retirement of instructions, and benchmarks from the 2006 SMT competition for the quantifier-free theory of uninterpreted functions over 32-bit bit-vectors. We used the latest version of MiniSat2 with simplification enabled to solve the resulting CNF problems [4]. The BAT time never exceeded 25 seconds for any benchmark, and MiniSat2 was given a timeout of 10 minutes. Experiments were run on an 2.4 GHz Intel Pentium 4 machine with a 512K cache and 1 GB of memory.

Scatter plots comparing our algorithm to the Tseitin and Jackson-Sheridan algorithms are given in Figure 3. Times reported are the total time taken for BAT and MiniSat2. Represented here are all benchmarks that took at least 0.1 second for Tseitin or Jackson-Sheridan to complete. Those taking less than a 0.1 second had comparable running times for all algorithms. Points above the diagonal indicate that our algorithm resulted in a faster overall solving time. Points along the top edge indicate time-outs for the competing algorithm. Based on these numbers, our algorithm clearly out-performs Tseitin and Jackson-Sheridan on almost all problems, often by orders of magnitude. Since this improvement is observed in the presence of CNF preprocessing (as performed by Minisat2), our view is that preprocessing techniques are not a satisfactory alternative to CNF translation [3]; rather these two approaches can be complementary.

Also, as noted earlier, despite the fact that our CNF translation is quadratic in the worst case, in practice it is faster that the linear-time Tseitin translation. The total time spent translating the benchmarks to CNF was 3,725 seconds for Tseitin and 3,409 seconds for our algorithm.

6 Conclusions

We presented the notion of NICE dags, a data structure that can be used to represent arbitrary circuits and outlined an algorithm that converts NICE dags to CNF. As our extensive experiments on over 8,000 benchmark problems show, our algorithm leads to very large efficiency gains in total SAT times over both an efficient variant of the widely used Tseitin translation algorithm and the Jackson-Sheridan algorithm, even in the presence of CNF preprocessing.

References

1. R. Brummayer and A. Biere. Local two-level and-inverter graph minimization without blowup. In *Proc. 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS '06)*, October 2006.
2. B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In T. Ball and R. B. Jones, editors, *Computer Aided Verification, CAV 2006*, volume 4144 of *LNCS*, pages 81–94, 2006.
3. N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.
4. N. Eén and N. Sörensson. MiniSat - a SAT solver with conflict-clause minimization. In F. Bacchus and T. Walsh, editors, *Posters of the 8th international Conference on Theory and Applications of Satisfiability Testing*, 2005.
5. P. Jackson and D. Sheridan. Clause form conversions for Boolean circuits. In H. H. Hoos and D. G. Mitchell, editors, *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004*, volume 3542 of *LNCS*, pages 183–198. Springer, 2004.
6. P. Manolios, S. K. Srinivasan, and D. Vroon. Automatic memory reductions for RTL-level verification. In *ICCAD 2006, ACM-IEEE International Conference on Computer Aided Design*. ACM, 2006.
7. P. Manolios, S. K. Srinivasan, and D. Vroon. BAT: The Bit-level Analysis Tool. 2006. Available from `http://www.cc.gatech.edu/~manolios/bat/`.
8. R. Nieuwenhuis and A. Oliveras. On SAT Modulo Theories and Optimization Problems. In A. Biere and C. P. Gomes, editors, *9th International Conference on Theory and Applications of Satisfiability Testing, SAT'06*, volume 4121 of *LNCS*, pages 156–169. Springer, 2006.
9. G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part2*, pages 115–125. Consultants Bureau, New York-London, 1962.
10. M. N. Velev. Efficient translation of boolean formulas to cnf in formal verification of microprocessors. In *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 310–315, 2004. IEEE Press.