Faster SAT Solving with Better CNF Generation*

Benjamin Chambers Panagiotis Manolios
Northeastern University
{bjchamb, pete}@ccs.neu.edu

Daron Vroon
General Theological Seminary of the Episcopal Church
daron.vroon@gmail.com

Abstract

Boolean satisfiability (SAT) solving has become an enabling technology with wide-ranging applications in numerous disciplines. These applications tend to be most naturally encoded using arbitrary Boolean expressions, but to use modern SAT solvers, one has to generate expressions in Conjunctive Normal Form (CNF). This process can significantly affect SAT solving times. In this paper, we introduce a new linear-time CNF generation algorithm. We have implemented our algorithm and have conducted extensive experiments, which show that our algorithm leads to faster SAT solving times and smaller CNF than existing approaches.

1. Introduction

SAT solving technology is now routinely used in a variety of areas ranging from hardware verification to artificial intelligence to computational biology. For many applications, users start by describing their problems using Boolean logic. However, to use most current SAT solvers, they then have to generate equisatisfiable CNF, the standard input format for most current SAT solvers. This is a non-trivial task, which is frequently performed by specially crafted algorithms and is often considered a technical contribution. More powerful CNF generation algorithms will provide a higher-level interface for users, lowering the bar to entry and making SAT technology more widely applicable.

For applications where SAT solvers are embedded in other tools, CNF generation is performed algorithmically, typically using the standard Tseitin algorithm. This is true even with verification tools that make extensive and essential use of SAT solving, such as Yices [8]. For such applications, the CNF generation process can significantly affect running times.

Finally, the performance of SAT solving technology seems to have reached a plateau, and further progress will most likely come from taking advantage of structure in problems before it gets obfuscated in the translation to CNF.

For all these reasons, we revisit CNF generation. Our contributions include:

- A new, linear-time CNF generation algorithm.
- NICE dags, a new data structure for representing Boolean expressions. NICE dags subsume And-Inverter Graphs (AIGs) [4] and are designed to provide better normal forms at linear complexity.
- An implementation of our algorithm in the NICESAT tool, which accepts as input both AIGs and NICE dags.
- Experimental evaluation, including the Hardware Model Checking Competition and SATRACE 2008 benchmarks, showing that NICESAT has faster SAT solving times and generates smaller CNF than existing tools.

2. Related Work

Related work can be divided into four categories: the representation of Boolean formulas, transformations applied to representations, CNF generation, and preprocessing of generated CNF before SAT solving.

There are numerous representations for propositional logic, including Reduced Boolean Circuits (RBCs) [2] and And-Inverter Graphs (AIGs) [12]. All of these representations are dags where internal nodes correspond to operations and leaf nodes correspond to input variables. In addition, they all employ structural-hashing to ensure that no duplicate nodes with the same operator and arguments appear in the dag. The representations differ in the operators they allow and the restrictions they impose on dags (whose purpose is to provide better normal forms).

*This research was funded in part by NASA Cooperative Agreement NNX08AE37A and NSF grants CCF-0429924, IIS-0417413, and CCF-0438871.

978-3-9810801-5-5/DATE09 © 2009 EDAA

Once a Boolean formula is converted into some machine representation, there are a variety of transformations that have been developed to simplify the dag. One approach is to replace small, local parts of the dag with smaller sub-dags [6, 9]. Typically such transformations are only applied when it can be shown they do not "hurt" sharing. Another set of techniques detect functionally equivalent sub-expressions in the graph and use one of the sub-expressions to eliminate the rest. An example is SAT sweeping [11].

Next comes CNF generation. Until recently, most CNF generation algorithms used in practice were minor variations of Tseitin's linear-time algorithm [17]. Tseitin's algorithm works by introducing new variables for internal nodes and constraining them so that equisatisfiable CNF is generated. Plaisted showed that one need only constraint an introduced variable in one direction when the corresponding node is used in only one polarity [16]. The AIGER tool includes a very efficient implementation of Tseitin's algorithm [4].

Boy de la Tour introduced a further improvement by showing how to compute the set of internal nodes which should have a variable introduced for them in order to minimize the number of clauses generated [7]. Another CNF generation algorithm is due to Velev, who identifies certain patterns arising in formulas from pipelined machine verification problems and provides CNF generation methods for these patters [18]. Jackson and Sheridan describe an algorithm that is a significant generalization of Velev's [10]. This algorithm converts RBCs to CNF and was designed to be simpler than the Boy de la Tour algorithm, while retaining many of its benefits. A key idea of the Jackson-Sheridan algorithm is to introduce a new variable for an argument to a disjunction only if that leads to fewer clauses. All of the CNF generation algorithms described in this paragraph have at least quadratic complexity, and the Jackson and Sheridan and Velev algorithms can generate output that is at least quadratic in the size of the input.[1]

In a previous short paper, we presented an informal overview of NICE dags, outlined aspects of a quadratic-time CNF generation algorithm and presented initial experimental results using BAT [13], a bounded model checker for a high-level, feature-rich, type-safe hardware description language that includes user-defined functions, arbitrary sized bit-vectors, and memories [14]. We also showed that the BAT algorithm generated simpler CNF and led to much faster times as compared with the Jackson and Sheridan algorithm. Since NICESAT outperforms BAT, it also handily outperforms the Jackson-Sheridan algorithm.

Another approach to CNF generation is based on technology mapping [9] and is implemented in ABC [3]. This algorithm computes a sequence of mappings, partial functions from AIG nodes to cuts of the graph, in an effort to minimize a heuristic cost function. CNF is then generated for the cuts of the nodes in the domain of the final mapping, using their irredundant sum-of-products representation.

[1] The claim in the Jackson and Sheridan paper that their algorithm is linear time is wrong.

3. NICE Dags

In this section we start by introducing *NICE expressions*, Boolean expressions over $\neg$, ite, iff, and $\wedge$ operators. We then introduce NICE dags, a structure-sharing representation for NICE expressions.

Boolean functions can be represented in numerous ways as a Boolean expression, and it is often helpful to use a formalism that limits the number of possible representations. At one extreme, we have unique, canonical representations (*e.g.*, BDDs), but at heavy computational cost. NICE expressions aim for a middle ground, where the computational costs are negligible, yet expressions are restricted in a way that allows us to efficiently do a fair amount of syntactic inference.

Definition 1. *Let $VARS$ be a set of Boolean variables. Then a* Negation, Ite, Conjunction, and Equivalence (NICE) expression *over $VARS$ is an expression that satisfies the following mutually recursive definition:*

$$
\begin{array}{rcl}
\textit{NiceExp} & ::= & \mathit{VARS} \\
& | & \textsc{True} \\
& | & \textsc{Not}\,(\textit{NiceExp}) \\
& | & \textsc{And}\,\left(\textit{NiceExp},\ \textit{NiceExp}^{+}\right) \\
& | & \textsc{Iff}\,(\textit{NiceExp},\ \textit{NiceExp}) \\
& | & \textsc{Ite}\,(\textit{NiceExp},\ \textit{NiceExp},\ \textit{NiceExp})
\end{array}
$$

In addition, we impose the following restrictions to obtain better normal forms:

- *If* TRUE *appears anywhere in a NiceExp, then the entire expression is either* TRUE *or* NOT (TRUE).
- *The argument to a* NOT *is not a negation.*
- *Neither of the arguments to an* IFF *are negated.*
- *Neither of the first* 2 *arguments to an* ITE *are negated.*
- *Arguments of expressions do not contain duplicates or negations of other arguments.*
- *Matching (see below).*

In practice, arguments are implemented as *pointers* or *edges* (we use both terms interchangeably), with negations appearing on the edge and all other operators (and variables) appearing as vertices. For the rest of the paper, we assume $VARS$ is fixed.

NICE dags are *structurally hashed* dags that are used to succinctly represent NICE expressions: if two sub-dags are structurally identical then they must be the same vertex. Given the correspondence between NICE dags and expressions, we sometimes use the terms "dag" and "expression" interchangeably. In practice, NICE dags are built bottom-up, using functions that create a new NICE dag by applying an operator to existing NICE dags. The functions `and`,

or, not, ite, iff, and impl all take NiceExps as arguments and return a structurally hashed NiceExp corresponding to the result of applying the appropriate logical operator to their arguments. To ensure uniqueness they make use of a global hash table, $GTAB$, using the arguments as keys.

Matching The and construction function also implements *matching*: it checks its arguments to determine if they encode an *ite* or *iff*. More specifically, matching succeeds when we try to construct a conjunction with two negated edges which point to binary conjunctions whose arguments are of the form v, v_1 and $\neg v, v_2$, respectively. When this occurs, we return $\texttt{not}\,(\texttt{ite}\,(v, v_1, v_2))$. The ite will get transformed into an *iff*, if it really encodes an equivalence. These techniques in conjunction with a collection of well-known rewrite rules are used to obtain normal forms. In practice, matching leads to non-trivial speedups for both CNF generation and SAT times.

4. CNF Generation

We present our algorithm for converting NICE dags to CNF in this section, after first presenting some of the key ideas upon which our algorithm depends.

We make essential use of *sharing* and *polarity*. Sharing occurs when a vertex has multiple parents in the NICE dag. This information is used by our algorithm when deciding whether to introduce new proxy variables.

The polarity of a path in a dag is *positive* (*negative*) if it contains an even (odd) number of negations. A vertex has *positive* (*negative*) polarity if there exists a positive (negative) path from the root of the dag to the vertex. Note that a vertex can have both positive and negative polarity. Procedures in our CNF generation algorithm often include an argument indicating polarity information. To simplify the presentation, we use the variable p, whose value is either + or −, to indicate polarity; also the operator $\overline{p}$ flips polarities.

Pseudo-Expansion Previous CNF generation algorithms deal with *ite* and *iff* by expanding such nodes away, say by using conjunctions and negations. We leave such nodes in the dag and define the function $\texttt{args}^p\,(e)$ which returns the *pseudo-expansion* of the node e in polarity p (see Figure 1). Leaving such nodes in the graph allows us to recognize $\texttt{args}^+\,(e)$ and $\texttt{args}^-\,(e)$ as semantic negations of each other even though they may not be syntactic negations. Also, expanding the node differently based on the polarity allows us to ensure we use the simplest expansion for the situation (the one that yields a conjunction of disjunctions). Pseudo-expansion leads to faster SAT solving times.

Counting Sharing Next, we annotate the DAG with information about the positive and negative sharing of each vertex using the code shown in Figure 2. For each vertex e, we record both the positive count ($\texttt{count}^+\,(e)$) and the negative count ($\texttt{count}^-\,(e)$). The positive count is the number of positive pointers to e that appear in a positive polarity and negative pointers to e that appear in a negative polarity. Similarly, $\texttt{count}^-\,(e)$ represents the number of positive pointers to e that appear in a negative polarity and the number of negative pointers to e that appear in a positive polarity. Note that if a pointer appears in both a positive and negative polarity it contributes to both the positive and negative counts of the vertex it points to.

$$
\begin{aligned}
\texttt{args}^+\,(\textsc{Ite}\,(x,\ y,\ z)) &= \{\texttt{impl}\,(x, y)\,, \texttt{impl}\,(\texttt{not}\,(x)\,, z)\}\\
\texttt{args}^-\,(\textsc{Ite}\,(x,\ y,\ z)) &= \{\texttt{impl}\,(x, \texttt{not}\,(y))\,,\\
&\qquad \texttt{impl}\,(\texttt{not}\,(x)\,, \texttt{not}\,(z))\}\\
\texttt{args}^+\,(\textsc{Iff}\,(x,\ y)) &= \{\texttt{impl}\,(x, y)\,, \texttt{impl}\,(y, x)\}\\
\texttt{args}^-\,(\textsc{Iff}\,(x,\ y)) &= \{\texttt{impl}\,(x, \texttt{not}\,(y))\,,\\
&\qquad \texttt{impl}\,(\texttt{not}\,(y)\,, x)\}
\end{aligned}
$$

Figure 1: Pseudo-expansions of *iff* and *ite*.

```
count-shares^p (e)
  if e = NOT (e') then e, p ← e', p̄
  if count^p (e) = 0 then
      if e = ITE (x, y, z) or e = IFF (x, y) then
          for all w ∈ args^p (e) do count-shares^+ (w)
      else if e = AND (W) then
          for all w ∈ W do count-shares^p (w)
  count^p (e) ← count^p (e) + 1
```

Figure 2: Annotating a NICE dag with sharing information.

All the counts are initialized to 0, which also allows us to detect whether a node has been previously visited. The code for $\texttt{count-shares}^p\,(e)$[2] performs a depth-first traversal of the dag rooted at e. The first conditional is used to check if we are visiting a negated edge, in which case we flip the polarity. A subtle case occurs when we visit the children of an *ite* or an *iff* vertex, in which case we look at the pseudo-expansion in the appropriate polarity and recursively call $\texttt{count-shares}^+\,(e)$ on these arguments. Why we call count-shares with positive polarity when $p = +$ should be clear. When $p = -$, note that the *ite* or *iff* appears in negative polarity, but $\texttt{args}^-\,(e)$ corresponds to the negation of e. The two negations cancel out and we call $\texttt{count-shares}^+\,(e)$.

Auxiliary Variables Efficient CNF generation requires the use of proxy variables. The code for introducing such variables appears in Figure 3. When $\texttt{var}^p\,(e)$ is called, we have the invariant that $\texttt{clauses}^p\,(e)$ contains CNF representing e in polarity p. The variable corresponding to e is x. If we previously introduced a variable for e in the opposite polarity, then x is the negation of that variable; otherwise, x is a new CNF variable. Surprisingly, the sign of the CNF variable can significantly affect SAT times. After experimenting with various options, we developed our *variable polarity heuristic*: we use polarity information to determine the sign of CNF variables, *i.e.*, if p is $+$, x is positive; oth-

[2] In our code, a "=" in a conditional returns true iff both the left and right sides have identical structure and it then binds variables in the obvious way.

```
var^p(e)
  if e = NOT(e') then e, p ← e', p̄
  if clauses^p̄(e) = {{y}} then x ← −y
  else x ← p(newvar())
  emit-clauses({{−x} ∪ c | c ∈ clauses^p(e)})
  clauses^p(e) ← {{x}}
```

Figure 3: Replacing a clause with a variable

```
cnf-main(e)
  pseudo-expand()
  count-shares^+(e)
  emit-clauses(cnf^+(e))
```

Figure 4: Main CNF generation function

erwise x is negative. We then constrain x so that it is equisatisfiable to e in polarity p, by introducing the constraint $x \rightarrow e$ (in CNF). This constraint appears in the final CNF we generate, so we write it to disk using `emit-clauses`; this allows us to free intermediate CNF which significantly reduces the memory and IO time required by our algorithm.

The Generation Algorithm The top-level CNF generation function, `cnf-main`(e) is shown in Figure 4. It generates CNF for a non-constant NICE dag, e, by calling the annotation functions we previously defined, and then invoking the recursive function `cnf`$^+(e)$, which is shown in figure Figure 5. The result of that call is a set of clauses corresponding to e; this set is written to disk by `emit-clauses`.

In brief, `cnf` works by annotating NICE dag vertices in a bottom-up fashion by calculating `clauses`$^p(e)$, the CNF representation of e in polarity p. These annotations are initially set to $null$. The first three **if** statements in `cnf` should be self-explanatory. If execution reaches the fourth **if**, then e is either a conjunction, ite, or iff. In either case, we generate CNF clauses which are stored in `clauses`$^p(e)$. We then use a heuristic to determine if a proxy variable should be introduced (via `var`$^p(e)$, as described previously). Our heuristic is encoded in `var-heur` and is explained later.

Finally, we explain how to compute `clauses`$^p(e)$, which depends on both the expression and the polarity. If e is an ite or a iff (the last **else**), we use the information we computed during pseudo-expansion to return (essentially) `cnf`$^+(\text{AND}(\text{args}^p(e)))$. Note that we always use positive polarity in the subcalls to `cnf` for the same reason we always used positive polarity when counting.

```
cnf^p(e)
  if e = NOT(e') then e, p ← e', p̄
  if clauses^p(e) ≠ null then return clauses^p(e)
  if e ∈ VARS then
      x ← newvar()
      clauses^+(e) ← {{x}}
      clauses^−(e) ← {{−x}}
      return clauses^p(e)
  if e = AND(W) and p = + then
      clauses^p(e) ← ⋃_{w∈W} cnf^+(w)
  else if e = AND(W) and p = − then
      clauses^p(e) ← dis(W)
  else
      clauses^p(e) ← ⋃_{w∈args^p(e)} cnf^+(w)
  if var-heur(count^p(e), clauses^p(e)) then var^p(e)
  return clauses^p(e)
```

Figure 5: Core of the CNF Generation Algorithm

```
dis(W)
  C ← {∅}
  for all e ∈ W do
      if dis-var-heur(C, clauses^−(e')) then var^−(e')
      C ← {b ∪ c | b ∈ clauses^−(e) ∧ c ∈ C}
  return C
```

Figure 6: Function for taking the disjunction of a set of vertices

```
var-heur(count, clauses)
  return count > 1 and ||clauses|| > 1

dis-var-heur(acc, e)
  return ||e||·|acc| + |e|·||acc|| > ||e|| + |acc| + |e| + ||acc|| or |e| > 4
```

Figure 7: Heuristics for minimizing the number of literals.

The last case to consider is when the vertex is a $\wedge$, in which case we either have a conjunction (if the polarity is positive) or a disjunction (if the polarity is negative). In the case of a conjunction, we take the union of all the clauses generated for the conjunction's arguments. In the case of a disjunction, we invoke the helper method `dis`(W), which is shown in Figure 6. This function begins by initializing an accumulator, C, to the singleton set containing the empty set. It then calls `cnf`$^-$ on each of the elements in $w \in W$, to ensure that `clauses`$^-(w)$ has been computed. Next, we iterate over the elements $w \in W$. Before constructing the disjunction $w \vee C$, we use the heuristic `dis-var-heur` to decide if we should introduce a variable for w.

Heuristics The heuristics used to decide when to introduce variables significantly affect the complexity of the CNF generation process and SAT times. While existing heuristics we know of are designed to minimize the number of clauses, our heuristics, shown in Figure 7, are designed to minimize the number of literals. Experimentally we have seen that this improves SAT times.

The heuristic for introducing variables when producing CNF is relatively simple: if the expression appears multiple times in the given polarity and currently takes more than a single literal to represent, we introduce a variable. We use double vertical bars to denote to the number of literals in a set of clauses, *i.e.*, $||D|| = \sum_{d \in D} |d|$. Single vertical bars denote length.

The heuristic for introducing variables for arguments of a disjunction consists of a local check to see if introducing a proxy for e will lead to an improvement and a global check to prevent the result of the disjunction from containing a large number of clauses. The combination of the two has been experimentally shown to produce better CNF and SAT times than either the global part or the local part alone.

k-Limited Merging Our algorithm often has to compute $c \cup d$, clause union. If we implement clause union by appending the two lists, we miss opportunities to detect clauses containing both x and $\neg x$ (*tautologies*) or multiple occurrences of x (*duplicates*). On the other hand, a full check is too computationally expensive. Our technique of k-limited merging occupies a middle ground: we maintain the invariant that the first k literals of a clause are sorted (by absolute value) and unique. A single $O(k)$ merge operation allows us to enforce the invariant and detect tautologies and duplicates. We ran experiments and were surprised to discover that any value of $k \geq 2$ is sufficient to pick up more than 99% of the opportunities for merging. In addition, detecting these tautologies and duplicates during the generation of CNF instead of during a post-processing step results in better SAT times because the simplifications interact in a virtuous way with our CNF-generation heuristics and algorithm. Surprisingly, merging also leads to a reduction in the CNF generation times due to reduced memory usage.

Theoretical Results

Theorem 1. *Given a NICE dag, e, `cnf-main`(e) emits a CNF formula that is equisatisfiable to e.*

Theorem 2. `cnf-main`(e) *has worst case running time $O(n)$, where n is the number of vertices in the dag representing e.*

Due to space limitations, proofs have been elided. We point out that obtaining Theorem 2 requires a certain amount of care with the data structures and heuristics used.

5. Experimental Evaluation

We implemented our algorithm in C++ and ran it on 850 different AIGs, including all 645 benchmarks from the 2008 Hardware Model Checking Competition (HWMCC '08) [5], 100 benchmarks from the structural SAT track of SAT-Race 2008 [1], and the 105 structural TIP benchmarks available from the AIGER [4] Website. The AIGs from the HWMCC were inducted using `aigbmc` with a value of $K = 45$ to produce structural benchmark problems.

We compared our algorithm with the Tseitin-inspired algorithm in AIGER, `aigtocnf`, and technology mapping, as implemented in ABC. These tools represent the state-of-the-art in CNF generation. Our experiments used the latest versions of AIGER and ABC, 20071012 and 70930, respectively. We used RSAT 2.01 [15], which was the overall fastest SAT solver we tried, but we also tested with PicoSat, MiniSat (with and without simplification) and observed similar results. The benchmarks were run on a cluster of Dual-core 2.0 GHz Intel Xeon EM64T machines each with either 8 or 16 GB of RAM. The results are summarized in Table 1. The NICESAT and AIGER transformations produced CNF for all the problems, while ABC failed on 31 problems, the numbers from which are not included in the table. The SAT solver was given a timeout of 1,000 seconds, and any benchmarks on which it timed out contributed 1,000 seconds to the SAT time (and total time) for the transformation. Of the instances that ABC produced CNF for, AIGER, ABC and NICESAT led to 103, 95 and 88 timeouts respectively. On the remaining 31 cases, AIGER and NICESAT led to 12 timeouts and 10 timeouts. It is also worth pointing out that NICESAT provides a higher-level interface than AIGER. Whereas AIGER accepts as input only structurally-hashed AIGs, NICESAT also accepts arbitrary Boolean expressions. This incurs a runtime cost during CNF generation, because, in contrast to AIGER, NICESAT does not assume that the input has been structurally-hashed. However, CNF generation time accounts for only a fraction of total running time and NICESAT is faster than AIGER and ABC on the benchmarks by 24K and 5K seconds respectively.

Table 1: Statistics for each SAT solver on benchmark problems

	AIGER	ABC	NICESAT
CNF Generation Time (s)	186	1224	310
SAT Time (s)	126K	107K	103K
Total Time (s)	127K	108K	103K
Variables	131M	44.2M	34.5M
Literals	888M	533M	424M
Clauses	380M	180M	129M

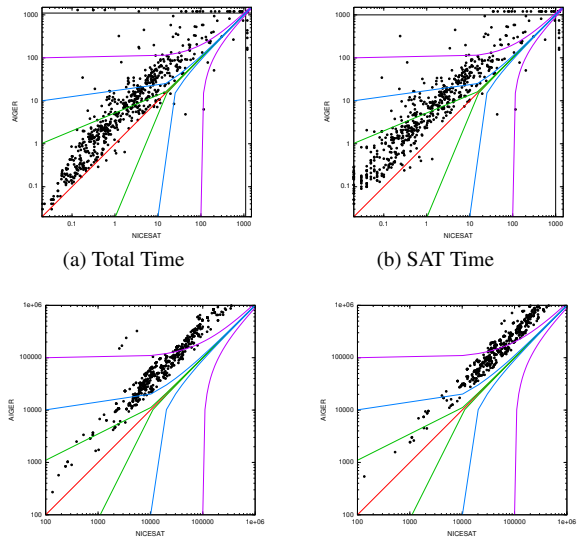
Log-scale scatter plots comparing NICESAT to AIGER and ABC are given in Figures 8 and 9. In all cases, smaller is better, so points above the diagonal denote benchmarks on which we get better results. Points on the top and right borders indicate failures to produce an answer within the time limit. The graphs show the total time to produce an answer (combining CNF generation and SAT), the SAT times, and the number of clauses and variables in the instance. We also plot the lines corresponding to a difference of 1, 10, $100, \ldots$ seconds, which aids in understanding the log-scale graphs.

In Figure 10 we show a graph of the time-limit versus the number of instances solved by the different tools. The figure shows that regardless of the time limit selected, NICESAT solves more problems than either ABC or AIGER.

Together, these experimental results show that we convincingly improve the state of the art in CNF generation.

6. Conclusions and Future Work

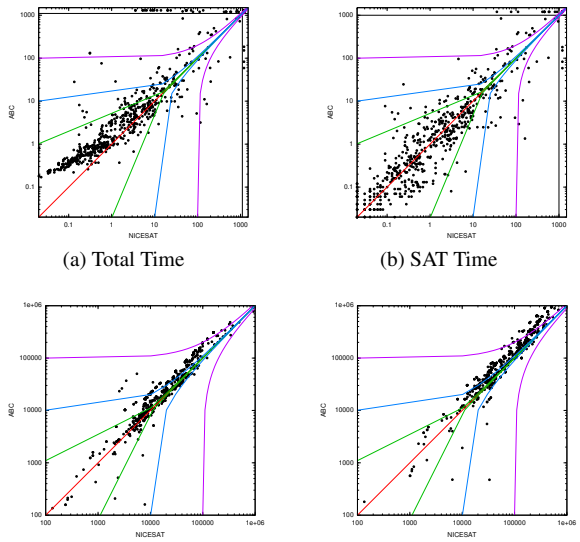
We presented NICE dags, a data structure that can be used to represent arbitrary circuits, and introduced a linear-time algorithm that converts NICE dags to CNF. We developed NICESAT, a C++ implementation that operates on NICE dags, AIGs, and Boolean expressions. By providing a higher-level interface, we lower the barrier to entry for new users of SAT technology. We have conducted extensive experiments which show that we generate smaller CNF and attain significant speed-ups in SAT solving times compared



(a) Total Time (b) SAT Time

(c) Variables (d) Clauses

Figure 8: Scatter plots comparing our algorithm to AIGER



(a) Total Time (b) SAT Time

(c) Variables (d) Clauses

Figure 9: Scatter plots comparing our algorithm to ABC

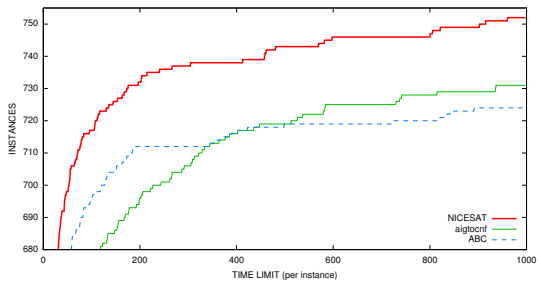


Figure 10: Time-limit vs. number of instances solved

with existing state-of-the-art tools. Novel techniques in our algorithm include k-matching, pseudo-expansion, our variable introduction heuristics, our variable polarity heuristic, and k-limited merging. Each of these techniques improves performance, and we plan on demonstrating this experimentally in the journal version of the paper. We also plan to develop additional techniques for improving CNF generation, reducing the size of the NICE dags, and adding support for quantification and bounded-model checking.

Finally, as CNF-based SAT solving is reaching a performance plateau, we need to look elsewhere for new improvements. What we have shown in this paper is that there is ample room for improvement in CNF generation. Therefore, we propose the study of linear-time algorithms for CNF generation.

References

[1] *SAT-Race 2008*. `http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/`.

[2] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic Reachability Analysis Based on SAT-Solvers. In *TACAS*, 2000.

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

[4] A. Biere. *AIGER Format and Toolbox*. `http://fmv.jku.at/aiger/`.

[5] A. Biere. *Hardware Model Checking Competition 2008*. `http://fmv.jku.at/hwmcc/`.

[6] R. Brummayer and A. Biere. Local Two-Level And-Inverter Graph Minimization without Blowup. In *MEMICS*, 2006.

[7] T. B. de la Tour. An Optimality Result for Clause Form Translation. *Journal of Symbolic Computation*, 1992.

[8] B. Dutertre and L. M. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proc. of CAV 2006*.

[9] N. Een, A. Mischenko, and N. Sorensson. Applying Logic Synthesis for Speeding Up SAT. In *SAT*, 2007.

[10] P. Jackson and D. Sheridan. Clause Form Conversions for Boolean Circuits. In *SAT*, 2004.

[11] A. Kuehlmann. Dynamic Transition Relation Simplification for Bounded Property Checking. *ICCAD*, 2004.

[12] A. Kuehlmann and F. Krohm. Equivalence Checking Using Cuts And Heaps. *Design Automation Conference*, 1997.

[13] P. Manolios, S. K. Srinivasan, and D. Vroon. BAT: The Bit-level Analysis Tool. `http://www.ccs.neu.edu/home/pete/bat`.

[14] P. Manolios and D. Vroon. Efficient Circuit to CNF Conversion. In *SAT*, 2007.

[15] K. Pipatsrisawat and A. Darwiche. RSat 2.0: SAT Solver Description. Technical Report D–153, UCLA, 2007.

[16] D. A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *JSAT*, 1986.

[17] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 1968.

[18] M. N. Velev. Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors. In *ASP-DAC*, 2004.