

The Optimality of a Fast CNF Conversion and its Use with SAT

Daniel Sheridan

University of Edinburgh, Kings Buildings, UK
d.j.sheridan@sms.ed.ac.uk

Abstract. Despite the widespread use and study of Boolean satisfiability for a diverse range of problem domains, encoding of problems is usually given to general propositional logic with little or no discussion of the conversion to clause form that will be necessary. In this paper we present a fast and easy to implement conversion to equisatisfiable clause form for Boolean circuits. Since the conversion is equivalent to that of Boy de la Tour it is optimal in the number of clauses produced. We present experimental results comparing this and other conversion procedures on BMC problems and conclude that the CNF conversion plays a large part in reducing the overall solving time.

1 Introduction

SAT solvers based on the DPLL procedure typically require their input to be in conjunctive normal form (CNF). Earlier papers dealing with encoding to SAT, particularly much of the planning literature, encode directly from the input representation to clause form. More recent encoding work makes little mention of CNF conversion. For example, Biere et al., proposing BMC [3], give an encoding to propositional logic only. Similarly, although the SNF encoding for BMC [6] discusses the clauses generated, the majority of the presentation is in general propositional logic. The microprocessor verification work of Velev includes a thorough analysis of improving the clause form generated [13], but the work is not immediately applicable to general propositional logic. Nevertheless, Velev is able to claim a speed up by a factor of 32 by altering the clause form conversion.

There is other evidence to motivate the study of clause form conversions for SAT. A reformulation [2] of a standard SAT benchmark problem using a more appropriate clause form representation is shown to reduce the solving time in modern solvers. Reformulating CNF problems automatically can have effective results but the procedures are frequently slow. By generating the improved clause form directly from the original problem, this may be avoided.

In the first-order logic domain, the CNF conversion problem was handled comprehensively by Boy de la Tour [4]. The algorithm given is impractical without the improvements by Nonnengart et al. [10], but the resulting algorithm is very fiddly to implement making it hard to be confident of a correct implementation.

In this paper we introduce a simple and easy to understand CNF conversion algorithm for propositional logic and prove that it is optimal with respect to the number of clauses. As its time complexity is linear, it represents a significant improvement over the (quadratic) Boy de la Tour algorithm. Of course, it is well known that problem size does not necessarily correspond to solving time in SAT, so we present some experimental results demonstrating the effect that our algorithm has on some BMC [3] problems.

1.1 Notation conventions

In an attempt to improve the clarity of the presentation, we use a number of conventions in our notation. Much of the work is concerned with both graphs and propositional logic, so we distinguish between *graph variables* representing vertices and edges given in italic capitals (X, Y) and *propositional variables* given in italic lower case (x, y); vertices are typically denoted V and edges E and this notation is significant in determining the type of a function. We will use the shorthand of referring to a subgraph by a single edge; the subgraph thus identified includes all of

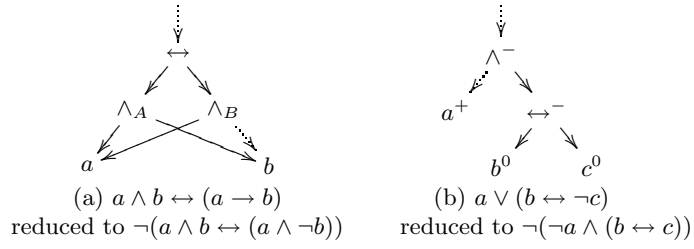


Fig. 1. Example RBCs showing vertex labelling

the descendants of the edge given, and such an edge is called the *root* of the subgraph and denoted T . Sets of vertices or edges are given in bold type (\mathbf{X} , \mathbf{Y}).

Where a function creates new propositional variables, these are given the name x_i where i is some identifier (typically a graph vertex). These variables are assumed to be unused in any other context.

2 Boolean Circuits

We present the algorithms in this paper using Boolean circuits, DAG representations of propositional logic which enable subformula sharing, even with different polarities, as a natural feature. A refinement called *reduced* Boolean circuits (RBCs) [1] restricts the structure of the graph to increase the amount of sharing. Using \mathbf{E} for the set of edges, \mathbf{V}_I for the set of internal (gate) vertices and \mathbf{V}_L for the set of leaf (variable) vertices, we can give the properties of RBCs as

An RBC is a DAG consisting of edges \mathbf{E} and vertices $\mathbf{V} = \mathbf{V}_I \cup \mathbf{V}_L$ where internal vertices \mathbf{V}_I representing operators, and leaf vertices \mathbf{V}_L representing variables. The following properties are required to hold and form the encoding of Boolean circuits as DAGs:

- Each $V \in \mathbf{V}_I$ consists of an operator $op(V) \in \{\wedge, \leftrightarrow\}$ and child edges $lt(V), rt(V) \in \mathbf{E}$
- Each $V \in \mathbf{V}_L$ contains a variable $var(V)$
- Each $E \in \mathbf{E}$ has a sign $sign(E) \in \{+, -\}$ and a target vertex $target(E) \in \mathbf{V}$

Negation is encoded into the edges by the *sign* attribute: $sign(E) = +$ indicates an unnegated edge and $sign(E) = -$ indicates a negated edge. Clearly, for full expressivity a formula in an RBC begins with an edge. To simplify the definitions in this paper we extend the set of properties on RBC vertices and edges with the inverse functions of *target*, and *lt* and *rt*:

$$in(V) = \{E \mid E \in \mathbf{E}, target(E) = V\} \quad source(E) = \begin{cases} V & \text{if } E = lt(V) \vee E = rt(V) \\ \text{undefined} & \text{otherwise} \end{cases}$$

RBCs have a number of other restrictions to reduce the number of representations of equivalent formulæ. Of interest here are: for all vertices, $lt(v) \neq rt(v)$; if $op(v) = \leftrightarrow$ then $lt(v)$ and $rt(v)$ are unsigned. The example graph in Figure 1a demonstrates the labelling of subgraphs that we will use, and the use of dashing for negated edges.

The eventual role of a conjunction vertex (as a conjunction or a disjunction) is determined by the number of negated edges appearing on a path connecting it to the root of the graph. In a manner similar to polarity for propositional logic we define in Figure 2 the number of *positive references* at a vertex V as the number of incoming edges from a positive polarity ancestor, and the number of *negative references* as the number of incoming edges from a negative polarity ancestor. These functions are defined with respect to the root edge of a formula so that only relevant edges are considered.

That is, $r_T^+(V)$ is increased by one for every positive incoming reference, but both positive and negative occurrences of equivalence reference their child vertices independently and hence increase the number of positive references separately; similarly for r_T^- . The conventional notion of polarity in propositional formulæ does not completely capture the semantics of r_T^+ and r_T^- even on RBCs structured as trees; nevertheless, we will use the following terms as a shorthand:

$$r_T^+(V) = \sum_{E \in \text{in}(V)} \begin{cases} 1 & \text{if } E = T, \text{sign}(E) = + \\ \min(r_T^+(\text{source}(E)), 1) & \text{if } \text{op}(\text{source}(E)) = \wedge, \text{sign}(E) = + \\ \min(r_T^-(\text{source}(E)), 1) & \text{if } \text{op}(\text{source}(E)) = \wedge, \text{sign}(E) = - \\ \min(r_T^+(\text{source}(E)), 1) + \min(r_T^-(\text{source}(E)), 1) & \text{if } \text{op}(\text{source}(E)) = \leftrightarrow \\ 0 & \text{otherwise} \end{cases}$$

Fig. 2. The positive references of vertex V with respect to root edge T ; r_T^- is defined similarly

- V has *positive polarity* if $r_T^+(V) \geq 1, r_T^-(V) = 0$
- V has *negative polarity* if $r_T^+(V) = 0, r_T^-(V) \geq 1$
- V has *zero polarity* if $r_T^+(V) \geq 1, r_T^-(V) \geq 1$

We use a system of annotations to the RBCs to indicate the polarity of a vertex, allowing us to draw diagrams of RBC fragments with properties that depend on the context of the fragment. Figure 1b shows the RBC corresponding to the formula $a \vee (b \leftrightarrow \neg c)$.

3 CNF Conversions

CNF conversions are usually defined on propositional formulæ without sharing. We therefore describe the CNF conversions over RBC *trees* rather than general graphs. We address the issue of sharing in Section 4.1. We cover the CNF conversions briefly here, but more detail is available in [12].

The *standard* CNF conversion is that obtained by the distributive properties of \wedge and \vee ; this results in an exponential increase in the size of the formula. It is easily applied to RBCs as a recursive descent of the tree, and we will write $\text{CNF}(T)$ for the conversion function.

Renaming is the introduction of new variables to represent the truth value of subformulæ. For example, a formula $(a \vee b) \wedge (c \vee d)$ may be converted more succinctly by the introduction of a variable $x_{c \vee d}$ thus: $((a \vee b) \vee x_{c \vee d}) \wedge (x_{c \vee d} \leftrightarrow (c \vee d))$. Given a positive or negative polarity subformula, the equivalence defining $x_{c \vee d}$ may be reduced to an implication with direction depending on the polarity. We write the polarity-sensitive renaming of an RBC tree as $\text{ren}(V, \mathbf{R})$ where $\mathbf{R} \subseteq \mathbf{V}_I$ is the set of vertices to rename. While this is sufficient to describe some CNF conversions, to define our conversion we require more direct control. We write $\text{CNF}_R(V, \mathbf{R}^+, \mathbf{R}^-)$ where \mathbf{R}^+ and \mathbf{R}^- are the sets of vertices for positive and negative polarity renaming respectively¹.

The *definitional* conversion $\text{DEF}(T) = \text{CNF}_R(T, \mathbf{V}_I, \mathbf{V}_I)$ renames every internal vertex both positively and negatively to avoid computing polarity (used in NuSMV [5], and in BCZChaff [7]); the *structure-preserving* [11] conversion, $\text{SP}(T) = \text{CNF}(\text{ren}(T, \mathbf{V}_I))$ takes into account the polarity of the subformulæ.

The *Boy de la Tour* conversion [4] is a method of choosing \mathbf{R} by computing the reduction in the total number of clauses brought by each renaming. Briefly, the conversion is based on the computation of the reduction in the overall clause size for renaming each vertex, performed top down. This is reduced to a function of the direct ancestors and descendents of the vertex. The conversion is shown to be optimal for formulæ without equivalences. See [12] for a description of the conversion applied to RBC trees. The main drawback of the approach is that the computation of the number of clauses at each vertex is based on exponentially growing functions: for even trivial BMC formulæ, the numbers overflow a 32-bit register. A later presentation of the algorithm by Nonnengart et al. [10] reduces the inequality to a number of case splits which can be solved by syntactic examinations of the surrounding graph. Unfortunately, these can become quite elaborate: the conditions for zero polarity formulæ require the evaluation of eight syntactic conditions in various combinations. We argue that it is prohibitively difficult to ensure a correct implementation of this method.

¹ It is not possible to write a convenient renaming function which takes positive and negative sets of vertices and returns an RBC as it could involve rewriting \leftrightarrow subgraphs; fortunately this does not affect our analysis.

Table 1. The renaming-compensated clause counting functions $p_r^+(T, \mathbf{R})$ and $p_r^-(T, \mathbf{R})$

	$p_r^+(V, \mathbf{R}^+, \mathbf{R}^-)$	$p_r^-(V, \mathbf{R}^+, \mathbf{R}^-)$
$V \in \mathbf{V}_L$	1	1
$V \in \mathbf{R}^+$	1	—
$V \in \mathbf{R}^-$	—	1
$op(V) = \wedge$	$p_r^+(lt(V), \mathbf{R}^+, \mathbf{R}^-) + p_r^+(rt(V), \mathbf{R}^+, \mathbf{R}^-)$	$p_r^-(lt(V), \mathbf{R}^+, \mathbf{R}^-)p_r^-(rt(V), \mathbf{R}^+, \mathbf{R}^-)$
$op(V) = \leftrightarrow$	$p_r^+(lt(V), \mathbf{R}^+, \mathbf{R}^-)p_r^-(rt(V), \mathbf{R}^+, \mathbf{R}^-) + p_r^-(lt(V), \mathbf{R}^+, \mathbf{R}^-)p_r^+(rt(V), \mathbf{R}^+, \mathbf{R}^-)$	$p_r^+(lt(V), \mathbf{R}^+, \mathbf{R}^-)p_r^+(rt(V), \mathbf{R}^+, \mathbf{R}^-) + p_r^-(lt(V), \mathbf{R}^+, \mathbf{R}^-)p_r^-(rt(V), \mathbf{R}^+, \mathbf{R}^-)$

$$\text{COMP}^+(T, E) = \text{COMP}^+(T, \text{target}(V))$$

$$\text{COMP}^-(T, E) = \text{COMP}^-(T, \text{target}(V))$$

$$\text{COMP}^+(T, V) = \begin{cases} \emptyset & \text{if } V \in \mathbf{V}_L, \text{ or} \\ \text{COMP}^+(T, lt(V)) \cup \text{COMP}^+(T, rt(V)) & \text{if } r_T^+(V) = 0, \text{ or} \\ \text{COMP}^+(T, lt(V)) \cup \text{COMP}^+(T, rt(V)) & \text{if } op(V) = \wedge, \text{ or} \\ \text{dis}^{+-}(V) \cup \text{dis}^{-+}(V) \cup \text{COMP}^+(T, lt(V)) \cup \text{COMP}^+(T, rt(V)) & \text{if } op(V) = \leftrightarrow \end{cases}$$

$$\text{COMP}^-(T, V) = \begin{cases} \emptyset & \text{if } V \in \mathbf{V}_L, \text{ or} \\ \text{COMP}^-(T, lt(V)) \cup \text{COMP}^-(T, rt(V)) & \text{if } r_T^-(V) = 0, \text{ or} \\ \text{dis}^{--}(V) \cup \text{COMP}^-(T, lt(V)) \cup \text{COMP}^-(T, rt(V)) & \text{if } op(V) = \wedge, \text{ or} \\ \text{dis}^{++}(V) \cup \text{dis}^{--}(V) \cup \text{COMP}^-(T, lt(V)) \cup \text{COMP}^-(T, rt(V)) & \text{if } op(V) = \leftrightarrow \end{cases}$$

$$\text{dis}^{xy}(V) = \left\{ \begin{array}{ll} \emptyset & \text{if } n_l n_r < n_l + n_r, \text{ or} \\ \{lt(V)\} & \text{if } n_l > n_r \\ \{rt(V)\} & \text{if } n_l \leq n_r \end{array} \right\} \text{ where } \left\{ \begin{array}{l} n_l = p_r^x(lt(V), \text{COMP}^+(T, lt(V)), \text{COMP}^-(T, lt(V))) \\ n_r = p_r^y(rt(V), \text{COMP}^+(T, rt(V)), \text{COMP}^-(T, rt(V))) \end{array} \right\}$$

Fig. 3. Renaming sets construction for the compact conversion

4 The Compact Conversion

We present a new clause form conversion, the *compact* conversion which computes the sets of renaming locally and bottom-up. Intuitively, we precompute r_T^+ and r_T^- for each vertex, then beginning with the leaves we work upwards through the graph computing the clauses which represent each vertex. A separate set of definitional clauses is maintained to define the variables used for renaming. At each vertex we consider the number of clauses it will generate based on whether a child is renamed: a disjunction $x \vee y$ is converted by either renaming an argument x and producing the clauses $\{\neg x\} \times \text{CNF}(y)$ or by simply computing $\text{CNF}(x) \times \text{CNF}(y)$, whichever results in the fewest clauses. Equivalences are handled as conjunctions of disjunctions so the same test can be used. Note that the decision to rename a vertex is made when considering the clauses generated by its *parent* vertex. The clauses for the positive and/or negative cases as required are then easily generated.

More precisely, we define the functions $\text{COMP}^+(T, V)$ and $\text{COMP}^-(T, V)$ in Figure 3 to give the set of positive and negative renamings respectively on the graph beginning at V . The auxiliary function $\text{dis}^{xy}(V)$ chooses the best child of V , if any, to rename given their signs, x and y . The renaming condition is computed on the tree after all vertices below the considered one have been renamed. To accommodate this we define a pair of functions (Table 1) which count the number of clauses produced by the graph beginning at vertex V after the renaming \mathbf{R} has been applied (the renaming analogue of the clause counting functions in [4]). In the implementation these values are the sizes of the clause sets representing V , so the conversion takes just $O(\mathbf{V})$ time.

Since we are targeting a SAT solver with this conversion, with its (assumed) exponential complexity in the number of variables, we choose to rename only if it *reduces* the number of clauses produced. In the case that the number of clauses is the same, the renaming is not performed. This is in contrast to the Boy de la Tour conversion, where the optimality analysis [4] is simplified by the zero-benefit renaming.

The optimality of the compact conversion is shown by comparing the renaming sets with that of the Boy de la Tour conversion. In [12] we show that for RBC trees, only the most immediate ancestor contributes to the renaming decision, and therefore the construction for the compact conversion is equivalent. Furthermore, by converting equivalences into graphs of conjunctions we show that the compact conversion is also optimal in this case.

4.1 RBCs with Sharing

Until now we have considered only RBCs structures as trees. As their main strength is in the sharing, we consider the changes required to make the compact conversion optimal for general RBCs. The basic approach for such a conversion is the repetition of multiply-referenced subgraphs with the alternative of renaming the subgraph in the same way as discussed above.

By a case analysis, we find that vertices with multiple inedges of the same polarity should be renamed if they generate more than one clause in that polarity; an exception is made if there are exactly two inedges with each referencing the vertex once. In this case renaming is not performed for size two. This special case is problematic as it extends the amount of information required about the graph beyond that available to the compact conversion. In fact, if we relax our requirement that renaming always reduces the number of clauses, this special case can be eliminated: renaming does not, in this case, increase the number of clauses generated. The resulting condition is $r_T^s(V) \geq 2$, $p^s(V) \geq 2$ $s \in \{+, -\}$.

Combining the multiple-reference renaming with the compact conversion requires some care to ensure that optimality is maintained. We must consider those vertices which are renamed during conversion to a tree that, had they not been renamed, would have resulted in the production of fewer clauses. This occurs only when $p_r^s(V, \text{COMP}^+(T, V), \text{COMP}^-(T, V)) = 1$ but $p^s(V) \geq 2$. The reverse situation (a renaming that should have been performed during tree construction was omitted) does not occur since $p_r^s(V, \text{COMP}^+(T, V), \text{COMP}^-(T, V)) \leq p^s(V)$. The optimal number of clauses is thus generated if the condition $p_r^s(V, \text{COMP}^+(T, V), \text{COMP}^-(T, V)) \geq 2$ is used in place of $p^s(V) \geq 2$. For this to be possible, the conversion to a tree must be performed bottom-up — the graph below V must already be a tree in order to compute the conversion condition required for converting V itself. An efficient implementation, running the two algorithms simultaneously, remains in $O(|\mathbf{V}|)$.

5 Evaluation and Conclusions

We have implemented the compact conversion in NuSMV [5] and compare it to the definitional conversion used by default. In addition, we can compare against an variation of the structure preserving conversion: we replace the condition $\text{dis}^{xy}(V)$ with one that always returns the larger child².

Table 2 lists the results and the timings with zChaff [9]. The problems are the standard DME benchmark and an industrial problem from the *Texas-97* benchmark suite (MSI) (see [6]), and two deadlock problems from [8] (Elevator and Mmgt). Where the SNF [6] encoding is used, this is indicated and we can see how the compact conversion helps to narrow the gap between this and the standard BMC encoding. Unsurprisingly, the compact conversion consistently generates fewer clauses and the solving times are also better in most cases, sometimes dramatically so. For the deadlock problems, however, the larger conversions perform significantly better. These problems are solvable, and it is likely to be a coincidence of variable ordering heuristics. Nevertheless it points a direction for further research into CNF conversions.

We conclude that the compact clause form conversion, which we have shown to be optimal in the number of clauses and a complexity class (linear versus quadratic) faster than the previously best known algorithm, also improves the solving time for several large benchmark problems.

² This is builds on SP in a number of ways: our implementation refuses to rename single clauses as it is never optimal; it also benefits from the optimal treatment of sharing.

Table 2. Benchmark results for three clause form conversions

Name	k	SAT	Definitional		SP		Compact	
			Clauses	Time (s)	Clauses	Time (s)	Clauses	Time (s)
DME (Priority 1, SNF)	13	Y	18 961	0.05	7 257	0.01	6 005	0.01
DME (Priority 1)	13	Y	22 043	0.05	8 189	0.01	6 630	0.01
DME (Priority)	52	Y	234 515	4.03	79 507	5.56	52 313	0.77
DME (Priority, SNF)	52	Y	75 121	2.63	28 785	1.32	23 789	0.47
DME (Access)	40	N	70 808	16.34	25 149	5.11	21 268	1.72
DME (Access, SNF)	40	N	58 884	14.14	22 484	2.63	18 640	1.13
MSI (Request A)	20	N	1 423 852	53.97	487 910	13.43	438 045	11.25
MSI (Request A, SNF)	20	N	1 422 861	80.28	487 915	20.49	438 066	13.64
MSI (Request B)	20	Y	1 423 187	174.2	488 011	40.23	438 169	49.80
MSI (Request B, SNF)	20	Y	1 424 359	174.7	488 288	29.37	438 423	23.11
Elevator 3	14	Y	230 104	1424.8	84 441	129.8	83 035	262.9
Mmgt 3	10	Y	44 556	28.2	16 770	794.4	16 116	376.9
Mmgt 4	12	Y	70 735	878.3	26 655	483.1	25 652	734.1

References

1. Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on SAT-solvers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 6th International Conference, TACAS'00*, volume 1785 of *Lecture Notes in Computer Science*, pages 411–425. Springer-Verlag, March 2000.
2. Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Principles and Practice of Constraint Programming — 9th International Conference, CP 2003*, Lecture Notes in Computer Science, 2003.
3. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems, 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, July 1999.
4. Thierry Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14:283–301, 1992.
5. Alessandro Cimatti, Marco Pistore, Marco Roveri, and Roberto Sebastiani. Improving the encoding of LTL model checking into SAT. In Agostino Cortesi, editor, *Third International Workshop on Verification, Model Checking and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*. Springer-Verlag, January 2002.
6. Alan Frisch, Daniel Sheridan, and Toby Walsh. A fixpoint based encoding for bounded model checking. In M D Aagaard and J W O'Leary, editors, *Formal Methods in Computer-Aided Design; 4th International Conference, FMCAD 2002*, volume 2517 of *Lecture Notes in Computer Science*, pages 238–254, Portland, OR, USA, November 2002. Springer-Verlag.
7. T. A. Juntilla. Boolean circuit tools (including BCZChaff). <http://www.tcs.hut.fi/~tjunttil/circuits>, May 2003.
8. Toni Jussila, Keijo Heljanko, and Ilkka Niemelä. BMC via on-the-fly determinization. In *First International Workshop on Bounded Model Checking*, July 2003.
9. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, Las Vegas, June 2001.
10. Andreas Nonnengart, Georg Rock, and Christoph Weidenbach. On generating small clause normal forms. In Claude Kirchner and Hélène Kirchner, editors, *Fifteenth International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 397–411. Springer-Verlag, 1998.
11. David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.
12. Daniel Sheridan. The optimality of a fast CNF conversion and its use with SAT. Technical Report APES-82-2002, APES Research Group, March 2004. Available from <http://www.dcs.st-and.ac.uk/~apes/apesreports.html>.
13. M. N. Velev. Efficient translation of Boolean formulas to CNF in formal verification of microprocessors. In *Asia and South Pacific Design Automation Conference (ASP-DAC '04)*, January 2004.