Journal on Satisfiability, Boolean Modeling and Computation 2 (2006) 1-25

Translating Pseudo-Boolean Constraints into SAT

Niklas Eén niklas@cadence.com
Cadence Berkeley Labs, Berkeley, USA.

Niklas Sörensson nik@cs.chalmers.se
Chalmers University of Technology, Göteborg, Sweden.

Abstract

In this paper, we describe and evaluate three different techniques for translating *pseudo-boolean* constraints (linear constraints over boolean variables) into *clauses* that can be handled by a standard SAT-solver. We show that by applying a proper mix of translation techniques, a SAT-solver can perform on a par with the best existing native pseudo-boolean solvers. This is particularly valuable in those cases where the constraint problem of interest is naturally expressed as a SAT problem, except for a handful of constraints. Translating those constraints to get a pure clausal problem will take full advantage of the latest improvements in SAT research. A particularly interesting result of this work is the efficiency of sorting networks to express pseudo-boolean constraints. Although tangential to this presentation, the result gives a suggestion as to how synthesis tools may be modified to produce arithmetic circuits more suitable for SAT based reasoning.

KEYWORDS: *Pseudo-boolean, SAT-solver, SAT translation, Integer Linear Programming*

Submitted October 2005; revised December 2005; published February 2006

1. Introduction

SAT-solvers have matured greatly over the last five years and have proven highly applicable in the *Electronic Design Automation* field. However, as SAT techniques are being embraced by a growing number of application fields, the need to handle constraints beyond pure propositional SAT is also increasing. One popular approach is to use a SAT-solver as the underlying decision engine for a more high-level proof procedure working in a richer logic; typically as part of an abstraction-refinement loop [11, 7]. Another common approach is to extend the SAT procedure to handle other types of constraints [15], or to work on other domains, such as finite sets or unbounded integers [20, 29].

In this paper we will study how a SAT-solver can be used to solve *pseudo-boolean* problems by a translation to clauses. These problems are also known as *0-1 integer linear programming* (ILP) problems by the linear programming community, where they are viewed as just a domain restriction on general linear programming. From the SAT point of view, pseudo-boolean constraints (from now on "PB-constraints") can be seen as a *generalization* of clauses. To be precise, a PB-constraint is an inequality on a linear combination of boolean variables: $C_0p_0+C_1p_1+\ldots+C_{n-1}p_{n-1} \geq C_n$, where the variables $p_i \in \{0,1\}$. If all constants C_i are 1, then the PB-constraint is equivalent to a standard SAT clause.

Stemming from the ILP community, pseudo-boolean problems often contain an *objective function*, a linear term that should be minimized or maximized under the given constraints.

Adding an objective function is also an extension to standard SAT, where there is no ranking between different satisfiable assignments.

Recently, a number of PB-solvers have been formed by extending existing SAT-solvers to support PB-constraints *natively*, for instance PBS [2], Pueblo [30], Galena [13], and, somewhat less recently, OPBDP [8], which is based on pre-Chaff [24] SAT techniques, but still performs remarkably well.

In this paper we take the opposite approach, and show how PB-constraints can be handled through translation to SAT without modifying the SAT procedure itself. The techniques have been implemented in a tool called MiniSat+, including support for objective functions. One of the contributions of our work is to provide a reference to which native solvers can be compared. Extending a SAT-solver to handle PB-constraints natively is, arguably, a more complex task than to reformulate the constraint problem for an existing tool. Despite this, a fair number of native solvers have been created without any real exploration of the limits of the simpler approach.

Furthermore, translating to SAT results in an approach that is particularly suited for problems that are *almost* pure SAT. Given a reasonable translation of the few non-clausal constraints, one may expect to get a faster procedure than by applying a native PB-solver, not optimized towards propositional SAT. Hardware verification is a potential application of this category.

2. Preliminaries

The satisfiability problem. A propositional logic formula is said to be in CNF, *conjunctive normal form*, if it is a conjunction ("and") of disjunctions ("ors") of literals. A literal is either x, or its negation $\neg x$, for a boolean variable x. The disjunctions are called *clauses*. The satisfiability (SAT) problem is to find an assignment to the boolean variables, such that the CNF formula evaluates to true. An equivalent formulation is to say that *each clause* should have at least one literal that is true under the assignment. Such a clause is then said to be *satisfied*. If there is no assignment satisfying all clauses, the CNF is said to be *unsatisfiable*.

The pseudo-boolean optimization problem. A *PB-constraint* is an inequality $C_0p_0 + C_1p_1 + \ldots + C_{n-1}p_{n-1} \geq C_n$, where, for all i, p_i is a literal and C_i an integer coefficient. A true literal is interpreted as the value 1, a false literal as 0; in particular $\neg x = (1 - x)$. The left-hand side will be abbreviated by *LHS*, and the right-hand constant C_n referred to as *RHS*. A coefficient C_i is said to be *activated* under a partial assignment if its corresponding literal p_i is assigned to True. A PB-constraint is said to be *satisfied* under an assignment if the sum of its activated coefficients exceeds or is equal to the right-hand side constant C_n. An *objective function* is a sum of weighted literals on the same form as an LHS. The pseudo-boolean optimization problem is the task of finding a satisfying assignment to a set of PB-constraints that minimizes a given objective function.

3. Normalization of PB-constraints

PB-constraints are highly non-canonical in the sense that there are many syntactically different, yet semantically equivalent, constraints for which the equivalence is non-trivial

to prove. For obvious reasons, it would be practical to get a canonical form of the PB-constraints before translating them to SAT, but to the best of our knowledge, no efficiently computable canonical form for PB-constraints has been found. Still, it makes sense to try to go some of the way by defining a *normal form* for PB-constraints. Firstly, it simplifies the implementation by giving fewer cases to handle; and secondly it may reduce some constraints and make the subsequent translation more efficient. In MiniSat+ we apply the following straightforward rules during parsing:

- $\leq$-constraints are changed into $\geq$-constraints by negating all constants.
- Negative coefficients are eliminated by changing p into $\neg p$ and updating the RHS.
- Multiple occurrences of the same variable are merged into one term $C_i x$ or $C_i \neg x$:
- The coefficients are sorted in ascending order: $C_i \leq C_j$ if $i < j$.
- Trivially satisfied constraints, such as "$x + y \geq 0$" are removed. Likewise, trivially unsatisfied constraints ("$x + y \geq 3$") will abort the parsing and make MiniSat+ answer Unsatisfiable.
- Coefficients greater than the RHS are *trimmed* to (replaced with) the RHS.
- The coefficients of the LHS are divided by their greatest common divisor ("gcd"). The RHS is replaced by "RHS/gcd", rounded upwards.

Example: $4x + 3y - 3z \geq -1$ would be normalized as follows:

$$\begin{array}{lll} 4x + 3y + 3\neg z & \geq & 2 \quad \text{(positive coefficients)} \\ 3y + 3\neg z + 4x & \geq & 2 \quad \text{(sorting)} \\ 2y + 2\neg z + 2x & \geq & 2 \quad \text{(trimming)} \\ y + \neg z + x & \geq & 1 \quad \text{(gcd)} \end{array}$$

Furthermore, after all constraints have been parsed, trivial conclusions are propagated. Concluding "$x =$True" is considered trivial if setting x to False would immediately make a constraint unsatisfiable. For example "$3x + y + z \geq 4$", would imply "$x =$True". Any assigned variable will be removed from all constraints containing that variable, potentially leading to new conclusions. This propagation is run to completion before any PB-constraint is translated into SAT.

Finally, MiniSat+ performs one more transformation at the PB level to reduce the size of the subsequent translation to SAT. Whenever possible, PB-constraints are split into a *PB part* and a *clause part*. The clause part is represented directly in the SAT-solver, without further translation. *Example:*

$$4x_1 + 4x_2 + 4x_3 + 4x_4 + 2y_1 + y_2 + y_3 \ \geq \ 4$$

would be split into

$$\begin{array}{lll} x_1 + x_2 + x_3 + x_4 + \neg z & \geq & 1 \quad \text{(clause part)} \\ 2y_1 + y_2 + y_3 + 4z & \geq & 4 \quad \text{(PB part)} \end{array}$$

where z is a newly introduced variable, not present in any other constraint. The rule is only meaningful to apply if the clause part contains at least two literals, or if the PB part is empty (in which case the constraint is equivalent to a clause).

For practical reasons, equality constraints ("$3x+2y+2z=5$") are often considered PB-constraints. To get a uniform treatment of inequalities and equalities, MiniSat+ internally represents the RHS of a PB-constraint as an interval $[lo, hi]$ (eg. "$3x + 2y + 2z \in [5, 5]$"). Open intervals are represented by $lo = -\infty$ or $hi = +\infty$. The parser puts some effort into extracting closed intervals from the input, so larger-than singleton intervals may exist. During normalization, closed intervals are first split into two constraints, normalized individually, and then, if the LHS:s are still the same, merged. However, for simplicity of presentation, only the "LHS $\geq$ RHS" form of PB constraints is considered in the remainder of the paper.

A slightly more formal treatment of PB-normalization can be found in [8], and some more general work on integer linear constraints normalization in [27]. We note that there exist 2^{2^n} constraints over n variables. A clause can only express 2^n of these, whereas a PB-constraint can express strictly more. To determine exactly how many more, which would give a number on the expressiveness of PB-constraints, and how to efficiently compute a canonical representation from any given constraint, is interesting future work.

4. Optimization – the objective function

Assume we have a PB minimization problem with an objective function $f(\mathbf{x})$. A minimal satisfying assignment can readily be found by iterative calls to the solver. First run the solver on the set of constraints (without considering the objective function) to get an initial solution $f(\mathbf{x_0}) = k$, then add the constraint $f(\mathbf{x}) < k$ and run again. If the problem is unsatisfiable, k is the optimum solution. If not, the process is repeated with the new smaller solution.

To be efficient, the procedure assumes the solver to provide an incremental interface, but as we only *add* constraints, this is almost trivial. A problem with the approach is that if many iterations are required before optimum is reached, the original set of constraints might actually become dominated by the constraints generated by the optimization loop. This will deteriorate performance.

The situation can be remedied by *replacing* the previous optimization constraint with the new one. This is sound since each optimization constraint subsumes all previous ones. In MiniSat+ this is non-trivial, as the constraint has been converted into a number of clauses, possibly containing some extra variables introduced by the translation. Those extra variables might occur among the *learned* clauses [15] of the SAT-solver, and removing the original clauses containing those variables will vastly reduce the pruning power of the learned clauses. For that reason MiniSat+ implements the naive optimization loop above as it stands; but as we shall see, other mechanisms prevent this from hurting us too much.

5. Translation of PB-constraints

This section describes how PB-constraints are translated into clauses. The primary technique used by MiniSat+ is to first convert each constraint into a single-output circuit, and

then translate all circuits to clauses by a variation of the Tseitin transformation [32]. The inputs of each circuit correspond 1-to-1 to the literals of the PB-constraints. The single output indicates whether the constraint is satisfied or not. As part of the translation to clauses, every output is forced to True. In MiniSat+, there are three main approaches to how a circuit is generated from a PB-constraint:

- Convert the constraint into a BDD.
- Convert the constraint into a network of adders.
- Convert the constraint into a network of sorters.

As circuit representation, we use RBCs (reduced boolean circuits) [1], meaning that the constant signals True and False have been eliminated by propagation, and that any two syntactically identical nodes have been merged by so-called *structural hashing*.

The structural hashing is important as all constraints are first converted to circuits, *then* to clauses. Constraints over similar sets of variables can often generate the same sub-circuits. Structural hashing also prevents the optimization loop from blowing up the SAT problem. Because the optimization constraints differ only in the RHS:s, their translations will be almost identical. Structural hashing will detect this, and few (or no) clauses will be added after the first iteration of the optimization loop. This is very important, as the objective function is often very large.

During the translation to clauses, extra variables are introduced in the CNF to get a compact representation. However, the goal of translating a PB-constraint into clauses is not only to get a compact representation, but also to preserve as many *implications* between the literals of the PB-constraint as possible. This concept is formalized in th CSP community under the name of *arc-consistency*. Simply stated, arc-consistency means that whenever an assignment could be propagated on the original constraint, the SAT-solver's unit propagation, operating on our *translation* of the constraint, should find that assignment too. More formally:

> **Definition.** Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ be a set of constraint variables, $\mathbf{t} = (t_1, t_2, \ldots, t_m)$ a set of introduced variables. A satisfiability equivalent CNF translation $\phi(\mathbf{x}, \mathbf{t})$ of a constraint $\mathcal{C}(\mathbf{x})$ is said to be *arc-consistent* under unit propagation *iff* for every partial assignment σ, performing unit propagation on $\phi(\mathbf{x}, \mathbf{t})$ will extend the assignment to σ' such that every unbound constraint variable x_i in σ' can be bound to either True or False without the assignment becoming inconsistent with $\mathcal{C}(\mathbf{x})$ in either case.

It follows directly from this definition that if σ is already inconsistent with $\mathcal{C}(\mathbf{x})$ (that is $\mathcal{C}(\mathbf{x})$ restricted to σ is empty), then unit propagation will find a conflict in $\phi(\mathbf{x}, \mathbf{t})$.

Although it is desirable to have a translation of PB-constraints to SAT that maintains arc-consistency, no polynomially bound translation is known. Some important special cases have been studied in [17, 4, 5, 6]. In particular, cardinality constraints, $x_1 + x_2 + \ldots + x_n \geq k$, can be translated efficiently while maintaining arc-consistency. Using the notion of arc-consistency, the translations of this paper can be categorized as follows:

- **BDDs.** Arc-consistent but exponential translation in the worst case.

- **Adders.** Not arc-consistent, $O(n)$ sized translation.
- **Sorters.** Not arc-consistent, but closer to the goal. $O(n \log^2 n)$ sized translation.

The parameter n here is the total number of digits[1] in all the coefficients. Translation through sorting networks is closer to the goal than adder networks in the sense that more implications are preserved, and for particular cases (like cardinality constraints), arc-consistency is achieved.

5.1 The Tseitin transformation

The first linear transformation of propositional formulas into CNF by means of introducing extra variables is usually attributed to Tseitin [32], although the construction has been independently discovered, in different variations, many times since then. For our purposes, a propositional formula is nothing but a single-output, tree-shaped circuit. The basic idea of the transformation is to introduce a variable for each output of a gate. For example, if the input signals of an And-gate have been given names a and b, a new variable x is introduced for the output, and clauses are added to the CNF to establish the relation $(x \leftrightarrow a \wedge b)$.

The final step of the transformation is to insert a unit clause containing the variable introduced for the single output of the circuit (or, equivalently, the top-node of the formula). It is easy to see that the models (satisfying assignments) of the resulting CNF are also models of the original propositional formula, disregarding the extra assignments made to the introduced variables.

As observed in [26], the transformation can be made more succinct by taking the *polarity* under which a gate occurs into account. A gate is said to occur *positively* if the number of negations on the path from the gate to the output of the circuit is even, and *negatively* if it is odd. Depending on the polarity, only the leftwards or rightwards part of the bi-implication $x \leftrightarrow \phi(a_1, a_2, \ldots, a_n)$, where ϕ is the gate being translated, needs to be introduced. When applying the Tseitin transformation to a *DAG-shaped* circuit,[2] a gate may occur both positively and negatively, in which case the full bi-implication must be established.

In the PB-translations to follow, the gate types listed below will be used. The inverter gate is not part of the list as negations can be handled by choosing the appropriate polarity x or $\neg x$ of a signal, without adding any clauses. In the listed clause representations, variable x always denotes the output of the gate under consideration. Clauses that need to be introduced for a positive/negative occurrence of a gate are marked with a $(+)/(-)$. For brevity, an over-line is used in place of "$\neg$" to denote negation:

- **And($\mathbf{a}_1$, $\mathbf{a}_2$, …, $\mathbf{a}_n$).** N-ary And-gate. Clause representation:

$$\begin{array}{ll} (-) & a_1 \wedge a_2 \wedge \ldots \wedge a_n \rightarrow x \\ (+) & \overline{a}_1 \rightarrow \overline{x},\ \overline{a}_2 \rightarrow \overline{x},\ \ \ldots\ , \overline{a}_n \rightarrow \overline{x} \end{array}$$

- **Or($\mathbf{a}_1$, $\mathbf{a}_2$, …, $\mathbf{a}_n$).** N-ary Or gate. Converted to And by DeMorgan's law.

- **Xor(a,b).** Binary Xor. Clause representation:

1. The radix does not matter for O-notation.
2. Directed Acyclic Graph. A circuit where outputs may feed into more than one gate.

$$\begin{array}{ll} (-) & a \wedge \overline{b} \rightarrow x \\ (-) & \overline{a} \wedge b \rightarrow x \end{array} \qquad \begin{array}{ll} (+) & a \wedge b \rightarrow \overline{x} \\ (+) & \overline{a} \wedge \overline{b} \rightarrow \overline{x} \end{array}$$

- **ITE(s,t,f).** If-then-else node with selector s, true-branch t, false-branch f, and output x. ***Semantics:*** $(s \wedge t) \vee (\neg s \wedge f)$. Clause representation:

$$\begin{array}{ll} (-) & s \wedge t \rightarrow x \\ (+) & s \wedge \overline{t} \rightarrow \overline{x} \end{array} \qquad \begin{array}{ll} (-) & \overline{s} \wedge f \rightarrow x \\ (+) & \overline{s} \wedge \overline{f} \rightarrow \overline{x} \end{array} \qquad \begin{array}{ll} (\text{red}-) & t \wedge f \rightarrow x \\ (\text{red}+) & \overline{t} \wedge \overline{f} \rightarrow \overline{x} \end{array}$$

  The two "red"-clauses are redundant, but including them will increase the strength of unit propagation.

- **FA_sum(a,b,c).** Output x is the "sum"-pin of a full-adder.[3] ***Semantics:*** $\textsc{Xor}(a, b, c)$. Clause representation:

$$\begin{array}{ll} (-) & \overline{a} \wedge \overline{b} \wedge \overline{c} \rightarrow x \\ (-) & \overline{a} \wedge b \wedge c \rightarrow x \\ (-) & a \wedge \overline{b} \wedge c \rightarrow x \\ (-) & a \wedge b \wedge \overline{c} \rightarrow x \end{array} \qquad \begin{array}{ll} (+) & a \wedge b \wedge c \rightarrow \overline{x} \\ (+) & a \wedge \overline{b} \wedge \overline{c} \rightarrow \overline{x} \\ (+) & \overline{a} \wedge b \wedge \overline{c} \rightarrow \overline{x} \\ (+) & \overline{a} \wedge \overline{b} \wedge c \rightarrow \overline{x} \end{array}$$

- **FA_carry(a,b,c).** Output x is the "carry"-pin of a full-adder. ***Semantics:*** $a + b + c \geq 2$. Clause representation:

$$\begin{array}{ll} (-) & b \wedge c \rightarrow x \\ (-) & a \wedge c \rightarrow x \\ (-) & a \wedge b \rightarrow x \end{array} \qquad \begin{array}{ll} (+) & \overline{b} \wedge \overline{c} \rightarrow \overline{x} \\ (+) & \overline{a} \wedge \overline{c} \rightarrow \overline{x} \\ (+) & \overline{a} \wedge \overline{b} \rightarrow \overline{x} \end{array}$$

- **HA_sum.** The "sum"-output of a half-adder. Just another name for Xor.

- **HA_carry.** The "carry"-output of a half-adder. Just another name for And.

For the ITE-gate, two redundant clauses (marked by "red") are added, even though they are logically entailed by the other four clauses. The purpose of these two clauses is to allow unit propagation to derive a value for the gate's output when the two inputs t and f are the same, but the selector s is still unbound. These extra propagations are necessary to achieve arc-consistency in our translation through BDDs (section 5.3). In a similar manner, propagation can be increased for the full-adder by adding the following clauses to the representation:[4]

$$\begin{array}{l} x_{carry} \wedge x_{sum} \rightarrow a \\ x_{carry} \wedge x_{sum} \rightarrow b \\ x_{carry} \wedge x_{sum} \rightarrow c \end{array} \qquad \begin{array}{l} \overline{x}_{carry} \wedge \overline{x}_{sum} \rightarrow \overline{a} \\ \overline{x}_{carry} \wedge \overline{x}_{sum} \rightarrow \overline{b} \\ \overline{x}_{carry} \wedge \overline{x}_{sum} \rightarrow \overline{c} \end{array}$$

3. A *full-adder* is a 3-input, 2-output gate producing the sum of its inputs as a 2-bit binary number. The most significant bit is called "carry", the least significant "sum". A half-adder does the same thing, but has only 2 inputs (and can therefore never output a "3").
4. Since we have split the full-adder into two single-output gates, we need to keep track of what sums and carries belong together to implement this.

In general, one wants the propagation of the SAT-solver to derive as many unit facts as possible. The alternative is to let the solver make an erroneous assumption, derive a conflict, generate a conflict-clause and backtrack—a much more costly procedure. If we can increase the number of implications made by the unit propagation (the *implicativity*[5] of the CNF) without adding too many extra clauses, which would slow down the solver, we should expect the SAT-solver to perform better. In general it is not feasible (assuming $P \neq NP$) to make all implications derivable through unit propagation by adding a sub-exponential number of clauses; that would give a polynomial algorithm for SAT. However, two satisfiability equivalent CNFs of similar size may have very different characteristics with respect to their implicativity. This partially explains why different encodings of the same problem may behave vastly differently. It should be noted that implicativity is not just a matter of "much" or "little", but also a matter of what cascading effect a particular choice of CNF encoding has. For a specific problem, some propagation paths may be more desirable than others, and the clause encoding should be constructed to reflect this.

5.2 Pseudo-code Conventions

In the pseudo-code of subsequent sections, the type "***signal***" represents either a primary input, or the output of a logical gate. The two special signals "True" and "False" can be viewed as outputs from pre-defined, zero-input gates. They are automatically removed by simplification rules whenever possible. For brevity, we will use standard C operators "&, |, ^" (also in contracted form "&=") to denote the construction of an And, Or, and Xor gates respectively. The datatype "***vec***" is a dynamic vector, and "***map***" is a hash-table. In the code, we will not make a distinction between word-sized integers and so-called *big-ints*. In a practical implementation, one may need to use arbitrary precision integers for the coefficients of the PB-constraints.

5.3 Translation through BDDs

A simple way of translating a PB-constraint into clauses is to build a BDD representation [12] of the constraint, and then translate that representation into clauses. The BDD construction is straightforward, and provided that the final BDD is small, a simple dynamic programming procedure works very efficiently (see Figure 3). In general, the variable order can be tuned to minimize a BDD, but a reasonable choice is to order the variables from the largest coefficient to the smallest. A rule of thumb is that important variables, most likely to affect the output, should be put first in the order. In principle, sharing between BDDs originating from different constraints may be improved by a more globally determined order, but we do not exploit that option.

Once the BDD is built, it can simply be treated as a circuit of ITEs (*if-then-else* gates) and translated to clauses by the Tseitin transformation. This is how MiniSat+ works. An example of a BDD translation, before and after reduction to the constant-free RBC representation, is shown in Figure 2. An alternative to using the Tseitin transformation, which introduces extra variables for the internal nodes, is to translate the BDD directly to clauses without any extra variables. This is done in [3], essentially by enumerating all paths

5. This terminology is introduced in [25], but the concept is studied in the CSP community as different *consistency* criteria (and methods to maintain them).

to the False terminal of the BDD. Yet another way to generate clauses from a decision diagram is to synthesize a multi-level And-Inverter graph, for instance by weak-division methods suggested by [23], and then apply the Tseitin transformation to that potentially much smaller circuit.

Analysis. BDDs can be used to translate cardinality constraints to a polynomially sized circuit. More precisely, $x_1+x_2+\ldots+x_n \geq k$ results in a BDD with $(n-k+1)\times k$ nodes, as illustrated in Figure 1. It is proven that in general a PB-constraint can generate an exponentially sized BDD [6]. Hence, in practice, a limit on the size must be imposed during the BDD construction. However, if the BDD is successfully built and translated by the Tseitin transformation, the resulting CNF preserves arc-consistency.

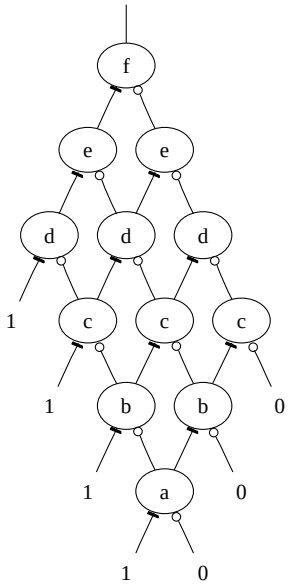


Figure 1. BDD for the cardinality constraint $a+b+c+d+e+f \geq 3$.

Proof: For simplicity, consider the ITE circuit without compaction by RBC rules. The result generalizes to RBCs in a straightforward manner. The BDD terminals are translated to literals fixed to True and False. Now, consider a translation of constraint $\mathcal{C}$ under the partial assignment σ. Let p_k be the unbound literal of $\mathcal{C}$ with the highest index. Observation: Since p_k is toggling the biggest coefficient, either $\mathcal{C},\sigma \models p_k$, or no implications exist. A smaller coefficient cannot be necessary if a bigger one is not. By the Tseitin transformation, the single output of the top-node is forced to True. For any node, if the BDD-variable ("selector") is bound, a True on the output will propagate down to the selected child. Because all literals p_i, where $i > k$, are bound and appear above p_k in the BDD (due to the variable order we use), True will propagate down to a unique BDD-node, call it N, branching on p_k. Furthermore, if a BDD-variable is bound, and the selected child is False, it will propagate upwards. Similarly, if both children are bound to False, it will propagate upwards by the redundant "red" clauses of our ITE translation (section 5.1). Inductively, if a BDD-node is equivalent to False under σ, the variable introduced for its output will be bound to False by unit propagation. Thus, if $\mathcal{C},\sigma \models p_k$, the variable introduced for the *false*-branch of N is False, and the output of N is True, which together will propagate p_k =True. □

Related Work. The PB-solver pb2sat [6] also translates PB-constraints to clauses via BDDs. The authors observe that the *false*-branch always implies the *true*-branch if the BDD is generated from a PB-constraint (or indeed any unate function). Using this fact, the Tseitin transformation can be improved to output two 3-clauses and two 2-clauses instead of six 3-clauses,[6] and still maintain arc-consistency.

The translation in [3] produces a minimal CNF under the assumption that no extra variables may be introduced. However, even polynomially sized BDDs can have an exponential translation to clauses under that assumption.

6. Although only half of the clauses will be instantiated if the BDD node occurs with only one polarity.

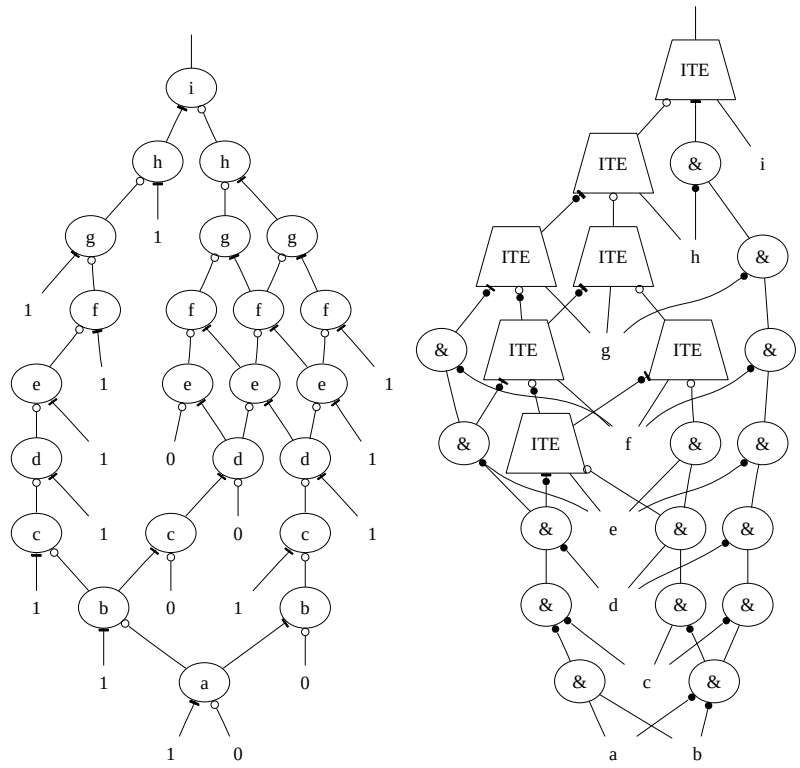


Figure 2. *The BDD, and corresponding RBC, for the constraint "$a + b + 2c + 2d + 3e + 3f + 3g + 3h + 7i \geq 8$".* The BDD terminals are denoted by "0" and "1". A circle ("o") marks the *false*-branch of a BDD-node, a dash ("-") marks the *true*-branch. In the RBC representation on the right, the unmarked edge is the selector (written, as customary, inside the node for the BDD). A solid circle ("•") denotes negation. Variables are ordered from the largest coefficient to the smallest, which has as consequence that no edge skips a variable in the BDD. Because the *false*-branch implies the *true*-branch of each node (assuming only positive coefficients, as in this example), no edge to the FALSE terminal comes from a *true*-branch.

5.4 Translation through Adder Networks

This section describes the translation of PB-constraints into clauses through half- and full-adders. First let us introduce some notation:

> **Definition.** A *k-bit* is a signal whose interpretation in the current context is the numeric value k. We will simply say that the bit *has* the value k.

The concept of k-bits is useful when manipulating bits representing numbers, or sums of numbers. *Example:* When adding two binary numbers "$a_3a_2a_1a_0$" and "b_1b_0", we call a_3 an *8-bit* (it contributes 8 to the sum if set). Signals a_1 and b_1 are the two *2-bits* of the sum.

```
signal buildBDD(vec⟨int⟩ Cs, vec⟨signal⟩ ps, int rhs,
                int size, int sum, int material_left, map⟨pair⟨int,int⟩, signal⟩ memo)
{
    if      (sum ≥ rhs)                  return TRUE
    else if (sum + material_left < rhs) return FALSE

    key = (size, sum)
    if (memo[key] == UNDEF) {
        size−−
        material_left −= Cs[size]
        hi_sum = sign(ps[size]) ? sum : sum + Cs[size]
        lo_sum = sign(ps[size]) ? sum + Cs[size] : sum
        hi_result = buildBDD(Cs, ps, rhs, size, hi_sum, material_left, memo)
        lo_result = buildBDD(Cs, ps, rhs, size, lo_sum, material_left, memo)
        memo[key] = ITE(var(ps[size]), hi_result, lo_result)
    }
    return memo[key]
}
```

Figure 3. *Building a BDD from the PB-constraint "*$\mathbf{Cs}\cdot\mathbf{ps} \geq rhs$*".* The functions used in the code do the following: "*sign*(p)" returns the sign of a literal (TRUE for $\neg x$, FALSE for x); "*var*(p)" returns the underlying variable of a literal (i.e. removes the sign, if any); and "ITE(cond,hi,lo)" constructs an *if-then-else* gate. The types ***vec***$\langle\cdot\rangle$" (a dynamic vector) and "***map***$\langle\cdot,\cdot\rangle$" (a hash table) are assumed to be passed-by-reference (the reasonable thing to do), whereas primitive datatypes, such as "***int***", are assumed to be passed-by-value. The BDD construction is initiated by calling "*buildBDD*()" with "size" set to the number of coefficients in the LHS, "sum" set to zero, "material_left" set to the sum of all coefficients in the LHS, and "memo" to an empty map.

Definition. A *bucket* is a bit-vector where each bit has the same value. A *k-bucket* is a bucket where each bit has the value k.

Definition. A *binary number* is a bit-vector where the bits have values in ascending powers of 2 (the standard representation of numbers in computers).

The translation of PB-constraints to clauses is best explained through an example. Consider the following constraint:

$$2a + 13b + 2c + 11d + 13e + 6f + 7g + 15h \geq 12$$

One way to enforce the constraint is to synthesize a circuit which adds up the activated coefficients of the LHS to a binary number. This number can then be compared with the binary representation of the RHS (just a lexicographical comparison). The addition and the corresponding buckets for the above constraint look as follows:

```
  00a0
  bb0b
  00c0
  d0dd
  ee0e
  0ff0
  0ggg
+ hhhh
```

Bucket:	Content:
1-bits:	[b,d,e,g,h]
2-bits:	[a,c,d,f,g,h]
4-bits:	[b,e,f,g,h]
8-bits:	[b,d,e,h]

The goal is to produce the sum as a binary number. It can be done as follows: Repeatedly pick three 2^n-bits from the smallest non-empty bucket and produce, through a full-adder, one 2^{n+1}-bit (the carry) and one 2^n-bit (the sum). The new bits are put into their respective buckets, possibly extending the set of buckets. Note that each iteration eliminates one bit from the union of buckets. When a bucket only has two bits left, a half-adder is used instead of a full-adder. The last remaining bit of the 2^n-bucket is removed and stored as the n^{th} output bit of the binary sum. It is easy to see that the number of adders is linear in the combined size of the initial buckets.

Pseudo-code for the algorithm is presented in Figure 5. We note that the buckets can be implemented using any container, but that choosing a (FIFO) queue—which inserts and removes from different ends—will give a balanced, shallow circuit, whereas using a stack—which inserts and removes from the same end—will give an unbalanced, deep circuit. It is not clear what is best for a SAT-solver, but MiniSat+ uses a queue.

For the lexicographical comparison between the LHS sum and the RHS constant, it is trivial to create a linear sized circuit. However, as we expect numbers not to be exceedingly large, we synthesize a comparison circuit that is quadratic in the number of bits needed to represent the sum (Figure 6). It has the advantage of not introducing any extra variables in the SAT-solver, as well as producing fewer (although longer) clauses.

Analysis. Adder networks provide a compact, linear (both in time and space) translation of PB-constraints. However, the generated CNF does not preserve arc-consistency under unit propagation. Consider the cardinality constraint $x_0 + x_1 ... + x_5 \geq 4$. The adder network synthesized for the LHS is shown in Figure 4. The RHS corresponds to asserting y_2. Now, assume x_0 and x_3 are both False. In this situation, the remaining inputs must all be True, but no assignment will be derived by unit propagation.

Another drawback of using adder networks is the *carry propagation problem*. Assume that x_0 and x_3 are now True instead. The two lower full-adders and the half-adder are summing 1-bits. Ideally, feeding two True-signals should generate a carry-out to the top full-adder, which is summing 2-bits. But because it cannot be determined which of the three outputs to the top full-adder is going to generate the carry, no propagation takes place.

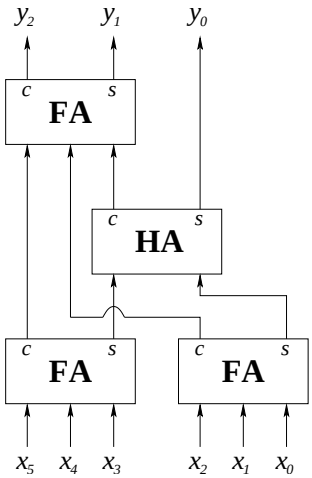


Figure 4. Adder circuit for the sum $x_0 + ... + x_5$.

Related Work. The algorithm described in this section is very similar to what is used for *Dadda Multipliers* to sum up the partial products [14]. A more recent treatment on multipliers and adder networks can be found

in [10]. Using adder networks to implement a linear translation of PB-constraint to circuits has been done before in [33, 3, 31], but the construction presented here uses fewer adders.

5.5 Translation through Sorting Networks

Empirically it has been noted that SAT-solvers tend to perform poorly in the presence of parity constraints. Because all but one variable must be bound before anything can propagate, parity constraints generate few implications during unit propagation. Furthermore, they tend to interact poorly with the resolution based conflict-clause generation of contemporary SAT-solvers. Because full-adders contain Xor (a parity constraint) we might expect bad results from using them extensively. In the previous section it was shown that translating cardinality constraints to adder networks does not preserve arc-consistency, which gives some theoretical support for this claim. Furthermore, the interpretation of a single bit in a binary number is very weak. If, for example, the third bit of a binary number is set, it means the number must be ≥ 8. But if we want to express ≤ 8, or ≥ 6 for that matter, a constraint over several bits must be used. This slows down the learning process of the SAT-solver, as it does not have the right information entities to express conflicts in.

To alleviate the problems inherent to full-adders we propose, as in [4], to represent numbers in *unary* instead of in binary. In a unary representation, all bits are counted equal, and the numerical interpretation of a bit-vector is simply the number of bits set to True. A bit-vector of size 8, for example, can represent the numbers $n \in [0, 8]$. Each such number will in the construction to follow be connected to a sorting network,[7] which allows the following predicates to be expressed by asserting a single bit: $n \geq 0$, $n \geq 1$, ..., $n \geq 8$, and $n \leq 0$, $n \leq 1$, ..., $n \leq 8$. Although the unary representation is more verbose than the binary, we hypothesize that the Xor-free sorters increase the implicativity, and that the SAT-solver benefits from having better information entities at its disposal for conflict-clause generation. The hypothesis is to some extent supported by our experiments, and we furthermore prove that the sorter-based translation of this section is arc-consistent for the special case of cardinality constraints.

To demonstrate how sorters can be used to translate PB-constraints, consider the following example:

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + 2y_1 + 3y_2 \geq 4$$

The sum of the coefficients is 11. For this constraint one could synthesize a sorting network of size 11, feeding y_1 into two of the inputs, y_2 into three of the inputs, and all the signals x_i into one input each. To assert the constraint, one just asserts the fourth output bit of the sorter.

Now, what if the coefficients of the constraint are bigger than in this example? To generalize the above idea, we propose a method to decompose the constraint into a number of interconnected sorting networks. The sorters will essentially play the role of adders on unary numbers. Whereas the adder networks of the previous section worked on binary numbers—restricting the computation to buckets of 1-bits, 2-bits, 4-bits, and so on for each power of 2— the unary representation permits us the use of any base for the coefficients, including a mixed radix representation. The first part of our construction will be to find a natural base in which the constraint should be expressed.

7. MiniSat+ uses *odd-even merge sorters* [9].

```
adderTree(vec⟨queue⟨signal⟩⟩ buckets, vec⟨signal⟩ result)
{
    for (i = 0; i < buckets.size(); i++) {
        while (buckets[i].size() ≥ 3) {
            (x,y,z) = buckets[i].dequeue3()
            buckets[i   ].insert(FA_sum(x,y,z))
            buckets[i+1].insert(FA_carry(x,y,z)) }

        if (buckets[i].size() == 2) {
            (x,y) = buckets[i].dequeue2()
            buckets[i   ].insert(HA_sum(x,y))
            buckets[i+1].insert(HA_carry(x,y)) }

        result[i] = buckets[i].dequeue()
    }
}
```

Figure 5. *Linear-sized addition tree for the coefficient bits.* The bits of "buckets[]" are summed up and stored in the output vector "result[]" (to be interpreted as a binary number). Each vector is dynamic and extends automatically when addressed beyond its current last element. The "***queue***" could be any container type supporting "*insert*()" and "*dequeue*()" methods. The particular choice will influence the shape of the generated circuit. Abbreviations "FA" and "HA" stand for full-adder and half-adder respectively.

```
// Generates clauses for "xs ≤ ys", assuming one of them has only constant signals.

lessThanOrEqual(vec⟨signal⟩ xs, vec⟨signal⟩ ys, SatSolver S)
{
    while (xs.size() < ys.size()) xs.push(False)        // Make equal-sized by padding
    while (ys.size() < xs.size()) ys.push(False)

    for (i = 0; i < xs.size(); i++) {
        c = False
        for (j = i+1; j < xs.size(); j++)
            c |= (xs[j] ^ ys[j])                        // "c = OR(c, XOR(xs[j], ys[j]))"
        c |= ¬xs[i] | ys[i]                             // Note, at this point "c ≠ False"
        S.addClause(c)
    }
}
```

Figure 6. *Compare a binary number "xs" to the RHS constant "ys".* One of the input vectors must only contain the constant signals True and False. If not, the function still works, except that "c" is not necessarily a clause when reaching "addClause()", so the call has to be replaced with something that can handle general formulas. *Notation:* the method "*push*()" appends an element to the end of the vector. In practice, as we put signals into the clause "c", we also apply the Tseitin transformation to convert the transitive fan-in ("all logic below") of those signals into clauses. Those parts of the adder network that are not necessary to assert "xs ≤ ys" will not be put into the SAT-solver.

Definition. A *base* $\mathbf{B}$ is a sequence of positive integers B_i, either finite $\langle B_0, \ldots, B_{n-1} \rangle$ or infinite.

Definition. A *number* $\mathbf{d}$ in a base $\mathbf{B}$ is a finite sequence $\langle d_0, \ldots, d_{m-1} \rangle$ of *digits* $d_i \in [0, B_i - 1]$. If the base is finite, the sequence of digits can be at most one element longer than the base. In that case, the last digit has no upper bound.

Definition. Let $tail(s)$ be the sequence s without its first element. The *value* of a number $\mathbf{d}$ in base $\mathbf{B}$ is recursively defined through:

$$value(\mathbf{d}, \mathbf{B}) \;=\; d_0 \;+\; B_0 \times value(tail(\mathbf{d}),\, tail(\mathbf{B}))$$

with the value of an empty sequence $\mathbf{d}$ defined as 0.

Example: The number $\langle 2, 4, 10 \rangle$ in base $\langle 3, 5 \rangle$ should be interpreted as $2 + 3 \times (4 + 5 \times 10) \;=\; 2 \times \mathbf{1} + 4 \times \mathbf{3} + 10 \times \mathbf{15} \;=\; 164$ (bucket values in boldface). Note that numbers may have one more digit than the size of the base due to the ever-present bucket of 1-bits. Let us outline the decomposition into sorting networks:

- Find a (finite) *base* $\mathbf{B}$ such that the sum of all the *digits* of the coefficients written in that base, is as small as possible. The sum corresponds to the number of inputs to the sorters we will synthesize, which in turn roughly estimates their total size.[8]
- Construct one sorting network for each element B_i of the base $\mathbf{B}$. The inputs to the i^{th} sorter will be those digits $\mathbf{d}$ (from the coefficients) where d_i is non-zero, *plus* the potential carry bits from the i-1^{th} sorter.

We will explain the details through an example. Consider the constraint:

$$a + b + 2c + 2d + 3e + 3f + 3g + 3h + 7i \geq 8$$

Assume the chosen base $\mathbf{B}$ is the singleton sequence $\langle 3 \rangle$, meaning we are going to compute with buckets of 1-bits and 3-bits. For the constraint to be satisfied, we must *either* have at least three 3-bits (the LHS ≥ 9), *or* we must have two 3-bits and two 1-bits (the LHS $= 8$). Our construction does not allow the 1-bits to be more than two; that would generate a carry bit to the 3-bits.

In general, let $\mathbf{d}^i$ be the number in base $\mathbf{B}$ representing the coefficient C_i, and $\mathbf{d}^{rhs}$ the number for the RHS constant. In our example, we have:

$$\begin{array}{lcll}
\mathbf{d}^0,\ \mathbf{d}^1 & = & \langle 1, 0 \rangle & \text{(digits for the } a \text{ and } b \text{ terms)} \\
\mathbf{d}^2,\ \mathbf{d}^3 & = & \langle 2, 0 \rangle & \text{(digits for the } c \text{ and } d \text{ terms)} \\
\mathbf{d}^4 .. \mathbf{d}^7 & = & \langle 0, 1 \rangle & \text{(digits for the } e, f, g, \text{ and } h \text{ terms)} \\
\mathbf{d}^8 & = & \langle 1, 2 \rangle & \text{(digits for the } i \text{ term)} \\
\mathbf{d}^{rhs} & = & \langle 2, 2 \rangle & \text{(digits for the RHS constant)}
\end{array}$$

8. In MiniSat+, the base is an approximation of this: the best candidate found by a brute-force search trying all prime numbers < 20. This is an ad-hoc solution that should be improved in the future. Finding the optimal base is a challenging optimization problem in its own right.

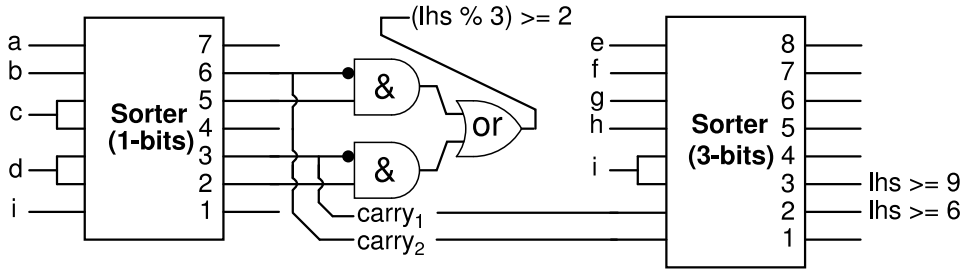


Figure 7. *Sorting networks for the constraint "$a+b+2c+2d+3e+3f+3g+3h+7i \geq 8$" in base* $\langle 3 \rangle$. Sorters will output 0:s at the top and 1:s at the bottom. In the figure, "lhs" is the value of the left-hand side of the PB-constraint, and "%" denotes the modulo operator. In base $\langle 3 \rangle$, 8 becomes $\langle 2, 2 \rangle$, which means that the most significant digit has to be either > 2 (which corresponds to the signal "lhs ≥ 9" in the figure), or it has to be $= 2$ (the "lhs ≥ 6" signal) and at the same time the least significant digit ≥ 2 (the "(lhs % 3) ≥ 2" signal). For clarity this logic is left out of the figure; adding it will produce the single-output circuit representing the whole constraint.

The following procedure, initiated by "*genSorters*(0, $\emptyset$)", recursively generates the sorters implementing the PB-constraint:

genSorters(n, *carries*)
- Synthesize a sorter of size: $(\sum_i d_n^i)+|carries|$. Inputs are taken from the elements of *carries* and the p_i:s of the constraint: signal p_i is fed to d_n^i inputs of the sorter.[9]
- Unless $n = |\mathbf{B}|$: Pick out every B_n:th output bit and put in *new_carries* (the outputs: $out[B_n]$, $out[2B_n]$, $out[3B_n]$ etc.). Continue the construction with *genSorters*($n+1$, *new_carries*).

Figure 7 shows the result for our example. Ignore for the moment the extra circuitry for modulo operation (the "lhs % $3 \geq 2$" signal). We count the output pins from 1 instead of 0 to get the semantics $out[n]$ = "at least n bits are set".

On the constructed circuit, one still needs to add the logic that asserts the actual inequality LHS $\geq$ RHS. Just like in the case of adder networks, a lexicographical comparison is synthesized, but now on the mixed radix base. The most significant digit can be read directly from the last sorter generated, but to extract the remaining digits from the other sorters, bits contributing to carry-outs have to be deducted. This is equivalent to computing the number of True bits produced from each sorter *modulo* B_i. Figure 8 shows pseudo-code for the complete lexicographical comparison synthesized onto the sorters. In our running example, the output would be the signal "(lhs ≥ 9) $\vee$ ((lhs ≥ 6) $\wedge$ (lhs % $3 \geq 2$))" (again, see Figure 7), completing the translation.

Analysis. In base $\langle 2* \rangle = \langle 2, 2, \ldots \rangle$, the construction of this section becomes congruent to the adder based construction described in the previous section. Sorters now work as adders because of the unary number representation, and as an effect of this, the carry propagation problem disappears. Any carry bit that can be produced *will* be produced

9. Implementation note: The sorter can be improved by using the fact that the carries are already sorted.

```
// Does the sorter output "out[1..size]" represent
   a number ≥ "lim" modulo "N"?
signal modGE(vec⟨signal⟩ out, int N, int lim)
  if (lim == 0) return True
  result = False
  for (j = 0; j < out.size(); j += N)
      result |= out[j + lim] & ¬out[j+N]
      // let out[n] = False if n is out-of-bounds
  return result
```

```
signal lexComp(int i)
  if (i == 0)
      return True
  else
      i−−
      out = "output of sorters[i]"
      gt = modGE(out, B_i, d_i^rhs + 1)
      ge = modGE(out, B_i, d_i^rhs)
      return gt | (ge & lexComp(i))
```

Figure 8. *Adding lexicographical comparison.* In the code, "sorters" is the vector of sorters produced by "*genSorters*()"—at most one more than $|\mathbf{B}|$, the size of the base used. The procedure is initialized by calling "*lexComp*(sorters.*size*())". Let $B_i = +\infty$ for $i = |\mathbf{B}|$ (in effect removing the modulus part of "*modGE*()" for the most significant digit). The $\mathbf{d}^{rhs}$ is the RHS constant written in the base $\mathbf{B}$. Operators "&" and "|" are used on signals to denote construction of And and Or gates.

by unit propagation. Moreover, the unary representation provides the freedom to pick any base to represent a PB-constraint, which recovers some of the space lost due to the more verbose representation of numbers.

Let us study the size of the construction. An odd-even merge sorter contains $n \cdot log\ n \cdot (1 + log\ n) \in O(n\ log^2 n)$ comparators,[10] where n is the number of inputs. Dividing the inputs between two sorters cannot increase the total size, and hence we can get an upper bound by counting the total number of inputs to *all* sorters and pretend they were all connected to the same sorter. Now, the base used in our construction minimizes the number of inputs, including the ones generated from carry-ins, so choosing the specific base $\langle 2* \rangle$ can only increase the number of inputs. In this base, every bit set to 1 in the binary representation of a coefficient will generate an input to a sorter. Call the total number of such inputs N. On average, every input will generate 1/2 a carry-out, which in turn will generate 1/4 a carry-out and so forth, bounding the total number of inputs, including carries, to $2N$. An upper limit on the size of our construction is thus $O(N\ log^2 N)$, where N can be further bound by $\lceil log_2(C_0) \rceil + \lceil log_2(C_1) \rceil + \ldots + \lceil log_2(C_{n-1}) \rceil$, the number of digits of the coefficient in base 2. Counting digits in another base does not affect the asymptotic bound.

By our construction, the cardinality constraint $p_1 + \ldots + p_n \geq k$ translates into a single sorter with n inputs, $p_1, \ldots, p_n$, and n outputs, $q_1, \ldots, q_n$ (sorted in *descending* order), where the k^{th} output is forced to True. We claim that unit propagation preserves arc-consistency. First we prove a simpler case:

Theorem: Assume exactly $n - k$ of the inputs p_i are set to 0, and that all of the outputs $q_1, \ldots, q_k$ are set to 1 (not just q_k), then the remaining unbound inputs will be assigned to 1 by unit propagation.

Proof: First note that the clauses generated for a single comparator locally preserve arc-consistency. It allows us to reason about propagation more easily. Further note that

10. A comparator is a two-input, two-output gate which sorts two elements. For boolean inputs, the outputs, called "min" and "max", corresponds to an And-gate and an Or-gate respectively.

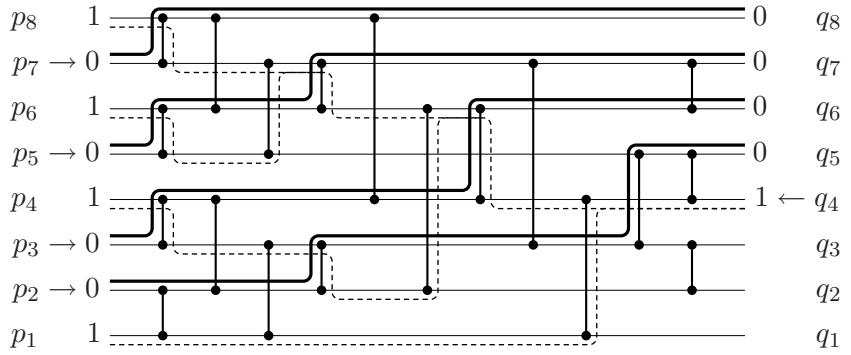


Figure 9. *Propagation trough an 8-sorter.* Forced values are denoted by an arrow; four inputs are set to *0* and one output is set to *1*. The thick solid lines show how the *0*s propagate towards the outputs, the thin dashed lines show how *1* propagate backwards to fill the unassigned inputs.

unit propagation has a unique result, so we are free to consider any propagation order. For brevity write *1*, *0*, and X for True, False and unbound signals.

Start by considering the forward propagation of *0*s. A comparator receiving two *0*s will output two *0*s. A comparator receiving a 0 and an X will output a *0* on the min-output and an X on the max-output. Essentially, the Xs are behaving like *1*s. No *0* is lost, so they must all reach the outputs. Because the comparators comprise a sorting network, the *0*s will appear contiguously at the highest outputs (see Figure 9).

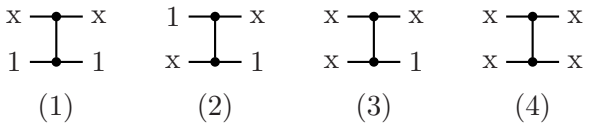
Now all outputs are assigned. Consider the comparators in a topologically sorted order, from the outputs to the inputs. We show that if both outputs of a comparator are assigned, then by propagation both inputs must be assigned. From this follows by necessity that the *1*s will propagate backwards to fill the Xs of the inputs. For each comparator, there are two cases (i) both outputs have the same value, in which case propagation will assign both inputs to that value, and (ii) the min-output of the comparator is *0* and the max-output is *1*. In the latter case, the *0* must have been set during the forward propagation of *0*s, so one of the comparator's inputs must be *0*, and hence, by propagation, the other input will be assigned to *1*. □

To show the more general case—that it is enough to set only the output q_k to *1*—takes a bit more work, but we outline the proof (Figure 9 illustrates the property):

- Consider the given sorter where some inputs are forced to *0*. Propagate the *0*s to the outputs. Now, remove any comparator (disconnecting the wires) where both inputs and outputs are *0*. For any comparator with both *0* and X inputs (and outputs), connect the X signals by a wire and remove the comparator. Do the same transformation if X is replaced by *1*. Keep only the network reachable from the primary inputs not forced to *0*. In essence: remove the propagation paths of the *0*s.
- Properties of the construction: (a) propagations in the new network correspond 1-to-1 with propagations in the original network, (b) the new network is a sorter.
- The original problem is now reduced to show that for a sorter of size k, forcing its top output[11] will propagate *1*s to all of its inputs (and hence all signals of the network).

11. Assume the orientation of Figure 9 for "up" and "down".

– Assume the opposite, that there exists an assignment of the signals that cannot propagate further, but which contains X. From the set of such assignments, pick a maximal element, containing as many *1*s as possible. An X can only appear in one of the following four configurations:



Assume first that (3) and (4) do not exist. In such a network, X behaves exactly as a *0* would do, which means that if we put *0*s on the X inputs, they would propagate to the outputs beneath the asserted *1* at the top, which violates the assumption that the network is sorting.

– One of the situations (3) or (4) must exist. Pick a comparator with such a configuration of Xs and set the lower input to *1*. A detailed analysis, considering the four ways Xs can occur around a comparator, shows that unit propagation can never assign a value to the X of the upper output of the selected comparator, contradicting the assumption of the assignment being maximal and containing X. To see this, divide the possible propagations into *forward propagations* (from inputs to outputs) and *non-forward propagations* (from outputs to inputs/outputs). A non-forward propagation can only propagate *1*s leftwards or downwards. A forward-propagation can never initiate a non-forward propagation. □

Unfortunately, arc-consistency is broken by the duplication of inputs, both to the same sorter and between sorters. Duplication within sorters can be avoided by using base $\langle 2* \rangle$, but duplication between sorters will remain. Potentially some improvement to our translation can re-establish arc-consistency in the general case, but it remains future work.

Related Work. Sorting networks is a well-studied subject, treated thoroughly by Knuth in [18]. For a brief tutorial on odd-even merge sorters, the reader is referred to [28]. In [4] an arc-consistent translation of cardinality constraints based on a $O(n^2)$ sized sorter (or "totalizer" by the authors terminology) is presented. The proof above shows that *any* sorting network will preserve arc-consistency when used to represent cardinality constraints. More generally, the idea of using sorting networks to convert non-clausal constraints into SAT has been applied in [19]. The authors sort not just bits, but words, to be able to more succinctly express uniqueness between them. Without sorting, a quadratic number of constraints must be added to force every word to be distinct, whereas the *bitonic sorter* used in that work is $O(n \log^2 n)$. In [16] a construction similar to the one presented in this section is used to produce a multiplier where the partial products are added by sorters computing on unary numbers. Although the aim of the paper is to remove Xors, which may have a slow implementation in silicon, the idea might be even better suited for synthesis geared towards SAT-based verification. Finally, in [34], the carry propagation problem of addition networks is also recognized. The authors solve the problem a bit differently, by preprocessing the netlist and extending the underlying SAT-solver, but potentially sorters can be used in this context too.

Small/medium ints. (577 benchmarks)

Solver	#solved-to-opt.
bsolo	187
minisat+	200
PBS4	166
Pueblo	194
sat4jpseudo	139
pb2sat+zchaff	150

Big integers (482 benchmarks)

Solver	#solved-to-opt.
bsolo	9
minisat+	26
PBS4	(buggy)
Pueblo	N/A
sat4jpseudo	3
pb2sat+zchaff	11

Figure 10. *PB-evaluation 2005.* Problems with objective function, solved to optimality.

6. Evaluation

This section will report on two things:

- The performance of MiniSat+ compared to other PB-solvers.
- The effect of the different translation techniques.

It is not in the scope of this paper to advocate that PB-solving is the best possible solution for a particular domain specific problem, and no particular application will be studied.

6.1 Relative performance to other solvers

In conjunction with the SAT 2005 Competition, a pseudo-boolean evaluation track was also arranged.[12] MiniSat+ participated in this evaluation together with 7 other solvers. Because not all solvers could handle arbitrary precision integers, the benchmark set was divided into *small*, *medium* and *big* integers. From the results, it seems that only the *big* category was problematic (greater than 30-bit integers), so we have merged the other two categories here. Not all problems had an objective function, so problems were also divided into *optimization* and pseudo-boolean *satisfiability* problems.

No optimization function (113 benchmarks)

Solver	#solved	(sat/unsat)
bsolo	44	(8/36)
minisat+	78	(35/43)
PBS4	89	(28/61)
Pueblo	103	(42/61)
sat4jpseudo	69	(17/52)
pb2sat+zchaff	78	(36/42)

Figure 11. *PB-evaluation 2005.* No objective function, just satisfiability (on small integers).

Out of the the 8 solvers, 3 was found to be unsound, reporting "UNSAT" for problems where other solvers found verifiable models. However, one of these solver, PBS4, seemed only to be incorrect for *big* integers, so we still consider this solver. But the other two, Galena and Vallst are excluded from our tables, as they have multiple erroneous "UNSAT" answers, even in the *small* integer category. The results of the remaining solvers can be found in Figure 10 and Figure 11. In the evaluation, MiniSat+ selected translation method using the following ad-hoc heuristic, applied to each constraint: *If the BDD translation is really compact, use that; otherwise, if the sorting network is not extremely large, use that; otherwise, fall back on adder networks (which are compact).*

12. http://www.cril.univ-artois.fr/PB05/ (see also [22, 21, 6, 2, 30, 13])

From the results we conclude that MiniSat+ and Pueblo were the two strongest solvers participating in the evaluation. Although it would be interesting to compare these solvers to commercial LP solvers such as CPlex, for practical reasons this has not been done—no LP solver was part of the evaluation, and CPlex requires a license.

6.2 Efficiency of different translation techniques

In this section, the three different translation techniques are evaluated on a random sample of benchmarks, drawn from the PB-evaluation set. Each benchmark is evaluated over 9 different parameters to MiniSat+:

- All constraints are translated using one and the same technique (3 choices).
- The objective function is translated using any of the techniques (3 choices).

The reason for treating the objective function separately, is that the optimization constraints generated from it are often very different from the problem constraints. The benchmarks were selected by the following procedure:

- Pick one of the 9 settings for MiniSat+.
- Pick one of the 1176 benchmarks from the evaluation set.
- If MiniSat+ could solve it within 10 minutes, keep it.
- Repeat until 40 benchmarks have been accumulated.

The procedure should give a reasonably unbiased benchmark set.[13] Run-times and translation sizes (relative to the PB-formulation) are presented in Figure 12. The experiments were carried out on a cluster of AMD Athlon XP 2800+ machines, each with 1 GB of RAM.

The results indicate that the translation through *adders* does not work well, either for the constraints or the objective function. This is particularly interesting as the adder-translation is the most compact one on average. For the objective function, it seems best to use the sorter-translation. If it is best combined with BDDs or sorters for the problem constraints cannot be concluded from the table, but there are instances where the result of using BDDs differ widely from the result of using sorters. In the table, the translation blow-up includes the objective function which can be dominating, as seen by comparing the results of optimization problems with the results of pure satisfiability problems.

Some remarks about the table: (i) The lower part (below the line) contains problems without objective function. As conversion of the non-existent objective does not affect the result, the same figures occur in three places, modulo timing fluctuation. (ii) The relative blow-up is given in logarithmic scale (the x of 10^x), and so $-\infty$ means that the problem was solved by the parser and pre-processor, producing zero clauses.

13. One benchmark was later detected as a duplicate, and therefore removed.

Constraints: **(Obj func.):**	Adders (Adder)	BDDs (Adder)	Sorters (Adder)	Adders (BDD)	BDDs (BDD)	Sorters (BDD)	Adders (Sorter)	BDDs (Sorter)	Sorters (Sorter)
afiro	293 (2.8)	299 (5.4)	**190** (3.3)	936 (4.4)	447 (5.4)	975 (4.8)	470 (2.9)	765 (5.4)	373 (3.5)
sc205	166 (2.4)	**3** (2.7)	86 (2.8)	166 (2.4)	**3** (2.7)	86 (2.8)	166 (2.4)	**3** (2.7)	86 (2.8)
bk4x3	7 (2.4)	4 (2.4)	10 (2.6)	21 (3.9)	32 (4.1)	15 (3.9)	9 (2.9)	6 (2.9)	**3** (2.9)
neos1	_ (1.2)	_ (1.2)	_ (0.8)	_ (1.4)	_ (1.5)	90 (1.3)	705 (1.2)	195 (1.2)	**23** (0.9)
neos20	8 (1.3)	3 (1.3)	14 (1.5)	9 (1.3)	**2** (1.3)	13 (1.5)	8 (1.3)	3 (1.3)	14 (1.5)
lseu	_ (2.6)	_ (3.0)	_ (2.9)	_ (4.2)	_ (4.2)	_ (4.2)	235 (3.1)	**203** (3.3)	331 (3.4)
misc03	30 (2.2)	_ (4.8)	33 (2.8)	**24** (2.1)	_ (4.8)	26 (2.8)	27 (2.1)	_ (4.8)	32 (2.8)
sample2	4 (2.6)	**3** (2.7)	6 (2.9)	7 (3.9)	8 (4.1)	5 (3.8)	21 (3.6)	17 (3.5)	17 (3.5)
stein45	20 (0.5)	25 (0.9)	21 (0.7)	16 (0.8)	18 (1.0)	16 (0.9)	20 (0.7)	20 (0.9)	**14** (0.7)
enigma	**5** (2.3)	122 (4.8)	9 (2.9)	**5** (2.3)	122 (4.8)	9 (2.9)	**5** (2.3)	123 (4.8)	9 (2.9)
noswot	_ (3.4)	119 (2.7)	_ (−∞)	_ (3.4)	131 (2.7)	_ (−∞)	_ (3.4)	**64** (2.7)	_ (−∞)
p0282	_ (2.4)	_ (2.5)	_ (2.7)	360 (2.1)	355 (2.3)	368 (2.6)	_ (3.1)	732 (3.3)	**281** (3.2)
vpm1	_ (2.4)	830 (3.4)	_ (2.9)	_ (2.4)	375 (3.4)	_ (2.9)	_ (2.4)	**176** (3.4)	_ (2.9)
sc50b	_ (2.6)	147 (2.7)	147 (2.8)	_ (2.6)	**39** (2.7)	143 (2.8)	_ (2.6)	108 (2.7)	139 (3.0)
neos8	_ (1.6)	359 (1.3)	**20** (−∞)	_ (1.7)	263 (1.8)	**20** (−∞)	_ (1.6)	266 (1.4)	**20** (−∞)
maros	12 (3.0)	**3** (3.2)	5 (3.2)	77 (4.5)	57 (4.5)	96 (4.6)	10 (3.1)	7 (3.3)	6 (3.4)
l152lav	_ (2.8)	429 (3.2)	704 (3.2)	_ (2.8)	110 (3.4)	**43** (3.6)	_ (2.8)	139 (4.4)	324 (4.7)
mod008	570 (3.6)	378 (3.6)	355 (3.7)	30 (5.4)	47 (5.4)	356 (5.9)	28 (4.6)	**20** (4.6)	67 (4.6)
clip-b	245 (0.9)	245 (0.9)	245 (0.9)	51 (1.4)	51 (1.4)	51 (1.4)	**6** (1.4)	**6** (1.4)	**6** (1.4)
hanoi5	**9** (0.5)	**9** (0.5)	**9** (0.5)	414 (2.6)	426 (2.6)	426 (2.6)	12 (1.2)	12 (1.2)	12 (1.2)
ii32b3	_ (0.5)	_ (0.5)	_ (0.5)	720 (1.8)	720 (1.8)	721 (1.8)	**25** (0.9)	**25** (0.9)	**25** (0.9)
ii32e4	_ (0.7)	_ (0.7)	_ (0.7)	_ (1.8)	_ (1.8)	_ (1.8)	**40** (0.8)	**40** (0.8)	**40** (0.8)
par16-3	**1** (0.7)	**1** (0.7)	**1** (0.7)	44 (2.5)	44 (2.5)	43 (2.5)	**1** (1.4)	**1** (1.4)	**1** (1.4)
ssa7552-159	_ (1.0)	_ (1.0)	_ (1.0)	_ (3.0)	_ (3.0)	_ (3.0)	**21** (1.6)	**21** (1.6)	**21** (1.6)
s4-4-3-2pb	_ (1.2)	_ (1.3)	_ (1.2)	_ (2.2)	942 (2.2)	393 (2.2)	131 (1.5)	320 (1.6)	**8** (1.4)
s4-4-3-9pb	26 (1.2)	150 (1.2)	23 (1.2)	454 (2.3)	684 (2.2)	153 (2.1)	14 (1.6)	61 (1.7)	**9** (1.8)
frb30-15-3	_ (0.1)	_ (0.1)	_ (0.1)	761 (0.7)	761 (0.7)	761 (0.7)	**164** (0.3)	**164** (0.3)	**164** (0.3)
frb35-17-5	_ (0.1)	_ (0.1)	_ (0.1)	_ (0.5)	_ (0.5)	_ (0.5)	811 (0.1)	811 (0.1)	**809** (0.1)
woodw	55 (−∞)	**54** (−∞)	55 (−∞)	55 (−∞)	**54** (−∞)	55 (−∞)	55 (−∞)	55 (−∞)	55 (−∞)
chnl10-11	60 (1.5)	**20** (1.4)	32 (1.5)	60 (1.5)	**20** (1.4)	32 (1.5)	60 (1.5)	**20** (1.4)	32 (1.5)
chnl10-20	**24** (1.8)	87 (1.5)	**24** (1.6)	**24** (1.8)	87 (1.5)	**24** (1.6)	**24** (1.8)	87 (1.5)	**24** (1.6)
fpga35-35	_ (1.0)	74 (0.9)	**3** (1.3)	_ (1.0)	75 (0.9)	**3** (1.3)	_ (1.0)	75 (0.9)	**3** (1.3)
fpga40-40	_ (1.0)	701 (0.9)	**2** (1.1)	_ (1.0)	700 (0.9)	**2** (1.1)	_ (1.0)	700 (0.9)	**2** (1.1)
22s-smv	3 (1.0)	**1** (0.7)	20 (1.2)	3 (1.0)	**1** (0.7)	20 (1.2)	3 (1.0)	**1** (0.7)	20 (1.2)
cache-inv12	36 (0.1)	**14** (0.0)	19 (0.2)	36 (0.1)	**14** (0.0)	19 (0.2)	36 (0.1)	**14** (0.0)	19 (0.2)
burch-dill	16 (0.8)	**3** (0.4)	16 (0.9)	16 (0.8)	**3** (0.4)	16 (0.9)	16 (0.8)	**3** (0.4)	16 (0.9)
ex-br-mem	**101** (0.9)	117 (0.6)	407 (1.1)	**101** (0.9)	116 (0.6)	407 (1.1)	**101** (0.9)	117 (0.6)	407 (1.1)
rf10	28 (0.4)	**15** (0.2)	18 (0.4)	28 (0.4)	**15** (0.2)	18 (0.4)	28 (0.4)	**15** (0.2)	18 (0.4)
tag10	96 (0.5)	**5** (0.3)	16 (0.6)	96 (0.5)	**5** (0.3)	16 (0.6)	96 (0.5)	**5** (0.3)	16 (0.6)
Solv. 100s:	18	17	22	18	19	23	23	24	**29**
Solv. 1000s:	23	29	28	26	33	33	31	**38**	37

Figure 12. *Runtime of* MiniSat+ *on a random selection of benchmarks.* The upper part contains optimization problems with an objective function; in the lower part are pure satisfiability problems. The numbers state runtime in seconds. A dash indicates timeout at 1000 seconds. The super-script numbers give the translation blow-up, written as log_{10}(clauses / pb-constraints). A value of "3" means each constraint was translated to 1000 clauses on average. The two top lines show what translation method was used for the constraints and objective function respectively. The two bottom lines show the total number of problems solved at a timeout of 100/1000 seconds.

7. Conclusions and Future Work

Coding integer arithmetic into boolean operations in a manner well suited for hardware implementation is a thoroughly studied topic. Contrary to this the focus of this paper lies on coding arithmetic, in particular the constructs present in PB-constraints, in ways that are suitable for a SAT-solver. One of the results shown herein is that the compact implementation of adder networks operating on binary numbers works poorly for SAT compared to the more verbose implementation using unary numbers. By change of representation SAT solving could be leveraged into PB solving with very reasonable results, which should have implications on, for example, how SAT-based circuit verification is carried out on designs containing arithmetic. Although making domain specific modification to a SAT-solver is an approach likely to outperform translation based methods in many cases, our belief is that translation is particularly well suited for the kind of problems where SAT-solvers are already successful. An example of such a problem might be finding error-traces in circuits with as many Xs on the inputs as possible. The pragmatics of the approach is also appealing, as encoding into SAT is often much easier than modifying the core solver.

A theoretical result of our study is a proven better bound on the smallest arc-consistent translation of cardinality constraints. Furthermore, our studies reinforce the common opinion that lack of carry propagation in adders are bad for SAT solving, and that more generally high implicativity or arc-consistency is desirable for the subcomponents of a SAT encoding. In the translation using BDDs, arc-consistency was achieved by adding redundant clauses to strengthen unit propagation, which further shows that "small" does not necessarily mean "good" when it comes to CNF encodings. In fact, an interesting branch of future research would be to study how a circuit can be partitioned into chunks that lend themselves to clausification with high implicativity or arc-consistency, such that (a) the encoding is still compact, and (b) the interaction of the components still preserves high implicativity. A more direct future work is to explore the freedom of base-selection in the translation using sorters, in particular by minimizing the number of duplicated inputs rather than the total number of inputs, to increase the implicativity.

8. Acknowledgments

The authors wish to express their gratitude to Alan Mishchenko, Armin Biere, Reiner Hähnle, Mary Sheeran and the reviewers for their careful reading and helpful suggestions for improvements of the manuscript. We would also like to thank Christian Szegedy who came up with the generalized version of the proof of arc-consistency of sorters, something we had hitherto only verified experimentally.

References

[1] P.A. Abdulla, P. Bjesse, and N. Een. **Symbolic Reachability Analysis Based on SAT-Solvers**. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)*, 2000.

[2] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. **PBS: A Backtrack Search Pseudo-Boolean Solver**. In *Symposium on the Theory and Applications of Satisfiability Testing (SAT'2002)*, 2002.

[3] F.A. Aloul, A. Ramani, I.L. Markov, and K.A. Sakallah. **Generic ILP versus Specialized 0-1 ILP: An Update**. In *Proceedings of International Conference on Computer Aided Design (ICCAD'2002)*, 2002.

[4] O. Bailleux and Y. Boufkhad. **Efficient CNF encoding of boolean cardinality constraints**. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming, CP 2003*, volume 2833. LNCS, 2003.

[5] O. Bailleux and Y. Boufkhad. **Problem encoding into SAT : the counting constraints case**. In *Proceedings of The Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'2004)*, 2004.

[6] O. Bailleux, Y. Boufkhad, and O. Roussel. **A Translation of Pseudo Boolean Constraints to SAT**. In *Journal on Satisfiability, Boolean Modeling and Computation (JSAT'06), issue 2*, pages 183–192, 2006.

[7] C.W. Barret, D.L. Dill, and A. Stump. **Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT**. In *Proc. of the 14^{th} Int. Conf. on Computer Aided Verification (CAV'02), LNCS 2404*, 2002.

[8] Peter Barth. **A Davis-Putnam Based Enumeration Algorithm for Linear pseudo-Boolean Optimization**. Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1995.

[9] K.E. Batcher. **Sorting Networks and their Applications**. In *Proceedings of AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.

[10] K.C. Bickerstaff, E.E. Swartzlander, and M.J. Schulte. **Analysis of Column Compression Multipliers**. In *Proceedings of 15^{th} IEEE Symposium on Computer Arithmetic (ARITH-15'01)*, 2001.

[11] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. v. Rossum, S. Schulz, and R. Sebastiani. **The MathSAT 3 System**. In *Conference on Automated Deduction (CADE-20)*, Springer Verlag, 2005.

[12] Randy E. Bryant. **Graph-Based Algorithms for Boolean Function Manipulation**. In *IEEE Transactions on Computers, C-35(8):677-691*, 1986.

[13] D. Chai and A. Kuehlmann. **A Fast Pseudo-Boolean Constraint Solver**. In *Proceedings of Design Automation Conference (DAC'03)*, pages 830–835, 2003.

[14] L. Dadda. **Some Schemes for Parallel Multipliers**. In *Alta Frequenza*, volume 34, pages 14–17, 1964.

[15] Niklas Een and Niklas Sörensson. **An extensible SAT solver**. In *Proceedings of the 6^{th} Int. Conference on Theory and Applications of Satisfiability Testing*, 2003.

[16] P.D. Fiore. **Parallel Multiplication Using Fast Sorting Networks**. In *IEEE Transactions on Computers, vol 48, no 6*, 1999.

[17] I.P. Gent. **Arc Consistency in SAT**. In *Proceedings of the Fifteenth European Conference on Artificial Intelligence (ECAI 2002)*, 2002.

[18] D. E. Knuth. **The Art of Computer Programming, volume 3: Sorting and Searching**. Addison Wesley, 1973.

[19] Daniel Kroening and Ofer Strichman. **Efficient Computation of Recurrence Diameters**. In *4th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *LNCS*, 2003.

[20] Cong Liu, Andreas Kuehlmann, and Matthew W. Moskewicz. **CAMA: A Multi-Valued Satisfiability Solver**. In *Int. Conf. on Computer Aided Design*, 2003.

[21] Vasco M. Manquinho and João Marques-Silva. **On Using Cutting Planes in Pseudo-Boolean Optimization**. In *Journal on Satisfiability, Boolean Modeling and Computation (JSAT'06), issue 2*, pages 191–210, 2006.

[22] Vasco M. Manquinho and Olivier Roussel. **The First Evaluation of Pseudo-Boolean Solvers**. In *Journal on Satisfiability, Boolean Modeling and Computation (JSAT'06), issue 2*, pages 97–136, 2006.

[23] S. Minato. **Fast Factorization Method for Implicit Cube Representation**. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 15, pages 377–384, 1996.

[24] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. **Chaff: Engineering an Efficient SAT Solver**. In *Proceedings of 12th International Conference on Computer Aided Verification*, volume 1855 of *LNCS*, 2001.

[25] Y. Novikov and R. Brinkmann. **Foundations of Hierarchical SAT-Solving**. In *6th Intl. Workshop on Boolean Problems, (extended ver.: ZIB-Report 05-38, 2005)*, 2004.

[26] D. Plaisted and S. Greenbaum. **A Structure-preserving Clause Form Translation**. In *Journal on Symbolic Computation 2*, 1986.

[27] William Pugh. **The Omega Test: a fast and practical integer programming algorithm for dependence analysis**. In *Supercomputing*, pages 4–13, 1991.

[28] Mary Sheeran. **Describing and reasoning about sorting networks**. In *slides from invited talk at the Nordic Workshop on Programming Theory (http://www.cs.chalmers.se/~ms/Turku.ppt)*, 2003.

[29] Hossein M. Sheini and Karem A. Sakallah. **A SAT-based Decision Procedure for Mixed Logical/Integer Linear Problems**. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'05)*, volume 3524 of *LNCS*, 2005.

[30] Hossein M. Sheini and Karem A. Sakallah. **Pueblo: A Hybrid Pseudo-Boolean SAT Solver**. In *Journal on Satisfiability, Boolean Modeling and Computation (JSAT'06), issue 2*, pages 157–181, 2006.

[31] Carsten Sinz. **Towards an Optimal CNF Encoding of Boolean Cardinality Constraints**. In *11th Int. Conf. on Principles and Practice of Constraint Prog.*, 2005.

[32] G. Tseitin. **On the complexity of derivation in propositional calculus**. *Studies in Constr. Math. and Math. Logic*, 1968.

[33] J.P. Warners. **A linear-time transformation of linear inequalities into conjunctive normal form**. In *Inf. Proc. Letters, 68, ISSN 0169-118X*, 1996.

[34] M. Wedler, D. Stoffel, and W. Kunz. **Arithmetic reasoning in DPLL-based SAT solving**. In *Design, Automation and Test in Europe Conference*, 2004.