Effective Preprocessing in SAT through Variable and Clause Elimination

Niklas Eén and Armin Biere

Chalmers University of Technology, Göteborg, Sweden.
Johannes Kepler University, Linz, Austria.

Abstract. Preprocessing SAT instances can reduce their size considerably. We combine variable elimination with subsumption and self-subsuming resolution, and show that these techniques not only shrink the formula further than previous preprocessing efforts based on variable elimination, but also decrease runtime of SAT solvers substantially for typical industrial SAT problems. We discuss critical implementation details that make the reduction procedure fast enough to be practical.

1 Introduction

The size of CNF formulas is often very large, particularly in the context of formal verification. In theory, a very large formula may be easy to solve and a small formula hard. However, in practice, it is often observed that the runtime of a SAT solver is very much related to the size of the input formula, at least when the formulas stem from the same set of problems.

This paper presents new techniques which reduce the size of a CNF formula in order to speed up overall SAT solving time. Our experiments on problems from industrial circuit verification show large speedups, not only compared to plain SAT solving, but also compared to related preprocessing techniques [21].

Modern SAT solvers use unit propagation and the pure literal rule in a preprocessing phase, as already described in the original DPLL algorithm [6]. More sophisticated techniques focus on deriving units, implications and equivalent literals [1,4,15,16,20]. Other similar techniques have been used in the context of ATPG, such as *recursive learning* [14], or in circuit verification, in particular *and-inverter graphs* [13] and *BDD-sweeping* [13]. The latter techniques have in common that they allow to restructure circuits but do not directly apply to CNF. Our approach is orthogonal to these techniques in the sense that it can be applied in addition to restructuring, after a CNF has been produced. Furthermore, from the CNF, individual clauses can be removed, an operation without correspondence in the circuit representation.

Preprocessing in SAT is a trade-off between the amount of reduction achieved and invested time. Light weight approaches such as [15] focus on fast preprocessing. Their running time is usually negligible compared to the overall solving time. On the other side of the spectrum lie techniques which in practice take considerable time. Examples are saturation of hyper binary resolution [1], k-saturation

for $k \geq 2$ in Stålmarck's method [20], or saturation of recursive learning for larger recursion depths. These techniques can only be applied if a huge benefit is expected—which is application domain depended—or if a time limit is enforced.

Recently, the rule of *elimination of atomic formulas* from [7], which eliminates variables from a CNF by *clause distribution*, has been reconsidered as a basis for symbolic DPLL in the ZDD based SAT solver ZRes [5], as a way to eliminate variables in the QBF solver Quantor [2], and, independently in the preprocessor NiVER [21]. Clause distribution is a light weight preprocessing technique, as long as a limit on the growth of the clause data base is enforced.

We extend [21] by three new techniques: subsumption, self-subsuming resolution, and variable elimination by substitution. This results in much higher reduction rates and faster SAT solving, as we show in our experiments. The description of the implementation of NiVER [21] stays on a very high level. We describe two implementation techniques for speeding up the process, based on (1) restricting the set of variables considered for elimination to *touched* variables, and (2) using *signatures* for fast subsumption checks. The latter has has also been used in [2], but the focus there is QBF, and no experimental results for preprocessing SAT instances are given. The optimizations allow us to keep the runtime small enough for a light weight approach.

Finally, we believe that preprocessing and encoding are two sides of the same coin. One way of speeding up SAT solving is to work on sophisticated CNF encoding algorithms such as [3,12,18,23]. We suggest, as an alternative, that the CNF is simplified after its generation, which is less application domain dependent. From a pragmatic point of view, it also eases the burden of developing good domain specific CNF encoders if the SAT solver is known to do a good job of reducing verbose CNF formulations. Furthermore, our simplification techniques are all resolution based and can therefore easily be incorporated in a solver with refutation generation. We leave it to future work to compare the two different approaches, particularly since the number of available CNF encoders and propositional non-CNF problems is currently rather small.

Generally, our simplification techniques can be applied in three different ways: (1) during preprocessing, (2) during SAT solving, e.g. at restart, or (3) between two incremental SAT problems. We will focus on applying simplification as a preprocessor, although a small study is included of an application in an incremental SAT problem.

To summarize, our contributions are the following. We extend NiVER [21] by subsumption, which was already discussed by one of the authors in the context of QBF [2]. Furthermore, we present two new techniques, self-subsuming resolution and variable elimination by substitution. Beside a compact description of the preprocessing algorithm itself we discuss two important low-level optimizations. Finally we show the effectiveness of these techniques as implemented in SatElite on a comprehensive set of industrial benchmarks.

2 Preliminaries

A CNF consists of a set of *clauses*, where each clause C is a set of literals. A literal is a boolean variable x or its negation $\overline{x}$.

Given two clauses $C_1 = \{x, a_1, \ldots, a_n\}$ and $C_2 = \{\overline{x}, b_1, \ldots, b_m\}$ the implied clause $C = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$ is called the *resolvent* of the two original clauses by performing *resolution* on the variable x. We write $C = C_1 \otimes C_2$. This notion can be lifted to sets of clauses. Let S_1 be a set of clauses which all contain x, S_2 a set of clauses which all contain $\overline{x}$. Then $S_1 \otimes S_2$ is defined as

$$S_1 \otimes S_2 = \{C_1 \otimes C_2 \mid C_1 \in S_1,\ C_2 \in S_2\}$$

The basic simplification technique in this paper, and also in NiVER [21], follows [7] and simply eliminates variables. In a given CNF, let S_x be the set of clauses in which x occurs, $S_{\overline{x}}$ be the set of clauses in which $\overline{x}$ occurs and define $S = S_x \cup S_{\overline{x}}$.

The elimination of a variable x in the whole CNF can be computed by pairwise resolving each clause in S_x with every clause in $S_{\overline{x}}$. The produced resolvents $S' = S_x \otimes S_{\overline{x}}$ replace the original clauses S containing x or $\overline{x}$, resulting in a satisfiability equivalent problem. We refer to this procedure as elimination by *clause distribution*, and count only non-trivial clauses as part of the result. A clause is trivial if it contains a variable and its negation.

In principle, the resolution operator $\otimes$ should have the resolution variable x as parameter. However, if clauses can be resolved with respect to different resolution variables, then all the resolvents will be trivial anyhow.

3 New Simplifications

In early experiments and also in the context of QBF [2] we observed that clause distribution produces many subsumed clauses. A clause C_1 is said to (syntactically) *subsume* C_2 if $C_1 \subseteq C_2$. A subsumed clause is redundant and can be discarded from the SAT problem. Particularly, a subsumed clause never needs to be part of a resolution proof of unsatisfiability.

We also observed that often similar clauses of a particular kind occur: one clause C_2 almost subsumes a clause C_1, except for one literal $\overline{x}$, which, occurs with the opposite sign in C_2. For instance, let $C_1 = \{x, a, b\}$, and $C_2 = \{\overline{x}, a\}$, then resolving on x will produce $C_1' = \{a, b\}$, which subsumes C_1. Thus after adding C_1' to the CNF, we can remove C_1, in essence eliminating one literal. In this case, we say that C_1 is *strengthened* by *self-subsumption* using C_2. This simplification rule is called *self-subsuming resolution*.

As we will show in the experimental section, adding these subsumption techniques to variable elimination through clause distribution gives huge benefits compared to [21].

If a circuit is encoded in CNF, typically using the Tseitin transformation [22], then many variables are actually *functionally dependent* on other variables, particularly those introduced for gate outputs. In previous work, this information has been used to restrict the set of decision variables in a SAT solver to

functionally *independent* variables [10,17]. We use the information to simplify the CNF, essentially extracting gates as in [17]. Output variables of gates are functionally dependent on input variables. If the following three clauses

$$\ldots \{x, \overline{a}, \overline{b}\}, \{\overline{x}, a\}, \{\overline{x}, b\} \ldots \tag{1}$$

are part of a CNF then the AND gate $x = (a \wedge b)$ can be extracted, showing that x is functionally dependent. We also call this equation a *definition* of x.

If x has a definition and is eliminated by clause distribution, many redundant resolvents are generated. By using the definition these clauses can be removed easily. Let G be the set of clauses used for extracting a *gate* with output x. Further recall that S_x is the set of clauses of S in which x occurs and similarly define G_x, and $G_{\overline{x}}$. Then the set S of all clauses with x or $\overline{x}$ can be partitioned into $S = G \cup R$, with $R \equiv S \backslash G$ the set of *remaining* clauses not used for extracting the gate. From $S = (G_x \cup R_x) \cup (G_{\overline{x}} \cup R_{\overline{x}})$ it follows that the set S' of all resolvents can be partitioned into $S' = S'' \cup G' \cup R'$ with

$$S'' = (R_x \otimes G_{\overline{x}}) \cup (G_x \otimes R_{\overline{x}}), \quad G' = G_x \otimes G_{\overline{x}}, \quad \text{and} \quad R' = R_x \otimes R_{\overline{x}}.$$

Furthermore, we have the following Theorem, which shows that S'' implies G' and R', allowing S' to be replaced by S''.

Theorem. $S'' \models G' \cup R'$

The proof follows by first noticing, as in [11], that G' contains only trivial clauses. All the resolvents in R' can be obtained through several resolution steps (linear in the width of the gate or by just one hyper resolution step [1]) from clauses in S''. Another view is to substitute in R all occurrences of x by its definition (x by $a \wedge b$ and $\overline{x}$ by $\overline{a \wedge b}$ in the example) and then apply the distributivity law to obtain a flat CNF.

As a result, in the elimination of a functional dependent variable the clauses in G' and R' do not have to be added, which always reduces the number of added resolvents. We call this simplification rule *variable elimination by substitution*. To continue the example in Eqn. (1) , let S be

$$\underbrace{\overset{1}{\{x, c\}}, \overset{2}{\{x, \overline{d}\}}}_{R_x}, \underbrace{\overset{3}{\{x, \overline{a}, \overline{b}\}}}_{G_x}, \underbrace{\overset{4}{\{\overline{x}, a\}}, \overset{5}{\{\overline{x}, b\}}}_{G_{\overline{x}}}, \underbrace{\overset{6}{\{\overline{x}, \overline{e}, f\}}}_{R_{\overline{x}}}$$

The resolvents are:

$$\overset{1\otimes4}{\{c, a\}}, \overset{1\otimes5}{\{c, b\}}, \overset{2\otimes4}{\{\overline{d}, a\}}, \overset{2\otimes5}{\{\overline{d}, b\}}, \overset{3\otimes6}{\{\overline{a}, \overline{b}, \overline{e}, f\}}\ (S'')$$

$$\overset{3\otimes4}{\{\overline{a}, \overline{b}, a\}}, \overset{3\otimes5}{\{\overline{a}, \overline{b}, b\}}\ (G') \qquad \overset{1\otimes6}{\{c, \overline{e}, f\}}, \overset{2\otimes6}{\{\overline{d}, \overline{e}, f\}}\ (R')$$

G' has only trivial clauses. Since trivial clauses are not counted, we have $|S'| = 7 > 5 = |S''|$. Replacing S with S'' results in a decrease of the number of clauses from 6 to 5, while the full clause distribution actually results in an increase from 6 to 7. Also note that the redundant clauses in R' can be obtained from S''

through two resolution steps each (actually by one hyper resolution step [1]): $1 \otimes 6 = (1 \otimes 4) \otimes ((1 \otimes 5) \otimes (3 \otimes 6))$ and $2 \otimes 6 = (2 \otimes 4) \otimes ((2 \otimes 5) \otimes (3 \otimes 6))$.

We also realized that subsumption sometimes removes clauses which could be used to extract a gate. For instance if the clause $C = \{\overline{a}, \overline{b}\}$ is added to the CNF in Eqn. (1), then the clause $\{x, \overline{a}, \overline{b}\}$ is removed and no AND gate can be extracted anymore. However, by one hyper resolution step, or two ordinary resolution steps, of C with the original two binary clauses the unit x can be derived, which, of course, simplifies the CNF even further. For all clauses C, we try to find binary clauses, that, if resolved with C in one hyper resolution step produce a unit. We call this simplification rule *hyper-unary-resolution*, similar to hyper-binary-resolution of [1].

4 Implementation

We present an implementation that should work for any clause based SAT solver, including those with an incremental SAT interface. In that context, the simplification can be applied between the different incremental SAT instances.

The techniques in this paper aim at simplifying a SAT problem by reducing its size. Variable elimination is applied greedily until no more improvement can be made to the clause database by a single elimination. Different notions of "improvement" can be used, and previous work [21] is focused on minimizing the number of *literal occurrences*. In our implementation we minimize the number of *clauses*. The rationale behind this is that propagation in a SAT solver is roughly proportional to the number of clauses, independent of their size.

4.1 Touched-lists

Subsumption and variable elimination interact, such that strengthening or removing a clause by (self-) subsumption can turn the elimination of a variable into an improvement, and eliminating a variable, which produces new clauses, might give new opportunities for subsumption.

In our implementation, subsumption and elimination are alternated until a fixed-point is reached. To make this efficient, it is important not to loop repeatedly over all clauses. Therefore, three sets are maintained, storing information about the modifications made to the clause database:

Touched (set of variables). A variable is added to this set if it occurs in a clause being added, removed, or strengthened. Initially all variables are "touched".

Added (set of clauses). When a clause is added to the SAT problem (e.g. by variable elimination), it is also added to this set. Initially all clauses are considered "added".

Strengthened (set of clauses). When a clause is strengthened (one literal is removed, either by self-subsumption or toplevel propagation[1]) it is added to this set. Initially the set is empty.

[1] Unit propagation performed under no assumptions, as opposed to during the search.

These sets are repeatedly *cleared* during the simplification procedure described in Sect. 4.3, then populated again as new clauses are produced during variable elimination, and while existing clauses are removed or strengthened by subsumption and self-subsumption. The algorithm terminates with all sets empty. In an incremental context—although not the focus of this paper—we note that new clauses can be added between SAT problems, populating *Added*, and that unit facts learned during the solving of one incremental SAT instance might remove or strengthen clauses, populating *Touched* and *Strengthened*.

4.2 Subsumption

The efficiency of subsumption is most important and is achieved by two implementation techniques. First, for each clause a 64-bit *signature* is stored [2]. The signature abstracts the set of literals of a clause in the following way: A hash function h maps literals to numbers 0..63, and the signature of a clause C is calculated as the bitwise OR of $2^{h(p)}$ over its literals $p \in C$. Then for each literal an *occur* list is maintained, pointing to all the clauses in which the literal occurs.

Now, *backward* subsumption, that is checking if a clause *subsumes* (as opposed to being *subsumed by*) some other clause in the database, can be implemented as follows:[2]

```
findSubsumed(Clause C)
  pick the literal p in C with the shortest occur list
  for each C' ∈ occur(p) do
      if (C≠C' && size(C)≤size(C') && subset(C, C'))
          add C' to result
  return result

subset(Clause C, Clause C')
  if (sig(C) & ~sig(C') ≠ 0) return FALSE
  else return result of iterating over C and C' in a
                 complete (expensive) subset test
```

This algorithm is very fast and allows backward subsumption to be applied *eagerly* to each added or strengthened clause. We rely on this fact in Sect. 4.3. Given a procedure for finding subsumed clauses, we can now define a method for using a clause C to strengthen other clauses by self-subsumption:

```
selfSubsume(Clause C)
  for each p ∈ C do
      for each C' subsumed by C[p := p̄] do
          strengthen(C', p̄)                    – remove p̄ from C'
```

For the clause $\{a, b, c\}$ this method would call *findSubsumed*() for $\{\overline{a}, b, c\}$, $\{a, \overline{b}, c\}$, $\{a, b, \overline{c}\}$, and strengthen any result returned. It should be noted that the order of strengthening matters, but is not optimized in our implementation.

[2] && denotes logical AND, & bitwise AND, and ~ bitwise negation.

4.3 The toplevel simplification method

We now state the main algorithm. The *post-conditions* are: (1) No opportunities remain for subsumption or self-subsumption. (2) No improvement can be made by eliminating a variable, unless the heuristic cut-off is used (see below). (3) The three sets *Touched*, *Added*, and *Strengthened* are empty.

```
simplify()
  do
      – SUBSUMPTION:
      S_0 = {set of clauses containing a literal occurring in
             some clause in Added}
      do
          S_1 = {set of clauses containing a literal occurring
                 negatively in some clause in Added}
              ∪ Added ∪ Strengthened
          clear Added and Strengthened
          for each C ∈ S_1 do selfSubsume(C)
          propagateToplevel()        – may strengthen/remove clauses
      while (Strengthened ≠ ∅)
      for each C ∈ S_0 not deleted do subsume(C)

      – VARIABLE ELIMINATION:
      do
          S = Touched ; clear Touched
          for each x ∈ S do maybeEliminate(x)
                  – eliminating variables will touch other variables
      while (Touched ≠ ∅)
  while (Added ≠ ∅)
```

The method *subsume*(C) removes any clause subsumed by C, and similarly *selfSubsume*(C) removes a literal from any clause that may be strengthened using C. The method *maybeEliminate*(x) removes x by clause distribution or substitution if the number of clauses is reduced. Finally, *propagateToplevel*() removes any satisfied clause or false literal permanently from the clause database, assigning variables and repeating the process if unit clauses are produced.

In the subsumption phase, two sets are computed: S_0 for standard subsumption, and S_1 for self-subsumption. Self-subsumption is applied first as it may render more (standard) subsumptions possible.

Because backward subsumption is eagerly applied to all added or strengthened clauses, the only candidates for being *subsumed* are the clauses of *Added*. Strengthened clauses cannot be subsumed as they now have fewer literals and were not subsumed before strengthening. A necessary condition for C to subsume C' is that C has at least one literal in common with C'. This motivates the definition of S_0.

Let "original clause" denote a clause not in *Added* or *Strengthened*. For self-subsumption (the set S_1) any added or strengthened clause can be used to re-

move literals from an original clause. Original clauses may self-subsume added clauses. This does not apply to strengthened clauses, since they have already been checked while still containing more literals. It remains to add to S_1 the original clauses that may strengthen a clause in *Added*. All the candidate clauses have to contain one literal $\overline{p}$ for some p in the added clauses.

4.4 Variable elimination

The variable elimination procedure relies on three readily implemented methods, which we state here without pseudo-code:

maybeClauseDistribute(x) eliminates x by clause distribution if the result has fewer clauses than the original (after removing trivially satisfied clauses).

findDefinition(x) returns either $x \leftrightarrow p_1 \vee p_2 \vee \ldots \vee p_n$ or $x \leftrightarrow p_1 \wedge p_2 \wedge \ldots \wedge p_n$ or NoDef. Unit information is also detected by hyper-unary-resolution and returned as $x \leftrightarrow$ True or $x \leftrightarrow$ False. Note that in general there may be many definitions. We use the shortest one and do not extract further information from this.

maybeSubstitute(*def*) takes the definition of a functionally dependent variable and substitutes each occurrence of the variable by its definition, provided this results in fewer clauses. Substituting a literal by a disjunction is unproblematic; substituting by a conjunction requires duplicating the destination clause for each literal of the conjunction, as explained in Sect. 3.

```
maybeEliminate( Var x)
  if (x assigned or has zero occurrences) return
  if (#occurs of x and ¬x are both > 10) return – heuristic cut-off
  def = findDefinition(x)
  if (def ≠ NoDef) maybeSubstitute(def)
  else             maybeClauseDistribute(x)
  if (x was eliminated)
      propagateToplevel()
      remove learned clauses with x      – for incremental SAT only
```

It was observed in an early implementation of the simplification procedure that on some problems the majority of time was spent on failed attempts to eliminate variables occurring frequently in both polarities. This is why these variables are heuristically excluded. The last line of the pseudo-code is only relevant in an incremental context; if variable elimination is applied during preprocessing, no learned clauses will exist.

4.5 Variable elimination related issues

The elimination of variables results in a partial model if the problem is satisfiable. Clauses removed during variable elimination must therefore be stored and used

to complete the model, if the full model is not needed. If not, removed clauses can simply be discarded.

Variable elimination also causes problems for the incremental SAT interface. Later extensions of the SAT instance might reintroduce eliminated variables, rendering the elimination unsound. Bringing back the removed clauses will solve the problem, but a simpler solution is to extend the solver interface to let the user explicitly prevent the elimination of selected variables.

5 Experimental results

The techniques presented in this paper were implemented in a tool SATELITE. It is downloadable together with the benchmarks and the result files used to produce the tables and diagrams of this section.[3] Three SAT solvers were used in our evaluation: (1) MINISAT v1.13 [8] with an improved conflict clause analysis [19]; (2) ZCHAFF version "Chaff II"; and (3) BERKMIN v5.61. The benchmarks were selected to be relevant for *circuit verification*. To get a relevant measure for the reduction achieved by our preprocessing techniques, unit clauses were removed by performing a toplevel propagation using MINISAT prior to benchmarking.

For our evaluation, two benchmark sets were created. The first set, referred to as "IBM Problems", is a subset of the huge BMC benchmark set made available by E. Zarpas at IBM.[4] The benchmark set is divided into directories, each containing BMC problems of different lengths generated from the same circuit with the same specification. Without any prior knowledge of the benchmarks, we randomly selected a subset of the directories resulting in 355 problems.

The second set, referred to as "Industrial Mix" contains a mix of hardware verification problems, obtained as follows: The available industrial problems of the SAT-2004 Competition were downloaded. Problems concerning graph coloring, set covering and planning problems were removed. Our focus is on circuit verification. We also removed *Miroslav Velev's* problems because SATELITE ran out of memory on some of them, which complicated benchmarking.[5] However, we note that the problems are already clausified in a smart way [23], which leaves little room for improvement by our methods. For the CNFs which SATELITE could preprocess, reduction rates of less than 5% were achieved, and no measurable speedup. This supports our hypothesis that our method is an *alternative* to producing optimized CNFs directly from the source problem.

Finally, we added 18 satisfiable and 18 unsatisfiable BMC problems used in [9], mainly from the *Texas'97 benchmarks*;[6] 18 unsatisfiable BMC problems

[3] www.cs.chalmers.se/~een/SatELite

[4] www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html

[5] The occurrence lists necessary for our preprocessing double the memory footprint. Although this is not a big issue, Velev's problems are among the largest that today's SAT solvers can handle. The current version of SATELITE has not been optimized for memory performance.

[6] www-cad.eecs.berkeley.edu/Respep/Research/vis/texas-97/

	Original			NiVER				SatELite as NiVER				Full SatELite			
Name	v	c	l	v	c	l	: t	v	c	l	: t	v	c	l	: t
6pipe	16	395	1157	15	393	1155	: 4.4	15	393	1155	: 2.2	**12**	**323**	**1018**	: 53.0
abp1-1-k31	15	48	124	8	34	98	: 0.6	8	33	94	: 0.3	**3**	**18**	**63**	: 1.2
barrel9	9	37	102	4	21	66	: 0.5	4	20	**65**	: 0.5	**2**	**16**	87	: 3.3
cache_10	227	880	2192	130	606	1680	: 20.6	92	417	1146	: 7.9	**29**	**178**	**748**	: 58.6
comb2	32	112	274	20	89	231	: 1.6	20	89	231	: 0.8	**3**	**18**	**63**	: 4.4
f2clk_40	28	80	186	10	44	125	: 1.4	7	32	90	: 0.5	**4**	**25**	**81**	: 1.2
fifo8_400	260	708	1602	69	301	859	: 13.6	42	164	451	: 6.5	**23**	**129**	**446**	: 11.2
guid-1-k56	99	307	758	45	193	553	: 3.9	44	189	540	: 3.1	**23**	**130**	**443**	: 8.0
ibm-03_k80	89	375	973	56	308	887	: 5.5	44	230	661	: 1.9	**28**	**190**	**629**	: 5.8
ibm-20_k45	91	373	945	46	281	832	: 6.7	41	250	725	: 2.1	**20**	**156**	**546**	: 7.0
ip50	66	215	513	34	148	398	: 5.1	12	50	**134**	: 1.6	**8**	**43**	139	: 4.2
longmult15	8	24	59	4	16	46	: 0.3	3	14	39	: 0.1	**1**	**9**	**28**	: 0.4
w08_14	120	425	1038	69	324	859	: 7.2	69	324	856	: 3.7	**34**	**220**	**688**	: 15.7

Table 1. *Size-reduction comparison with* NiVER. "v", "c", "l" denote the number of variables, clauses, and literals in thousands respectivly. Times "t" are in seconds as provided by the Unix command "time", and include parsing and writing the result file. "SatELite as NiVER" uses no subsumption and has the same heuristic as NiVER for variable elimination (enforce fewer literals). It shows that SatELite can mimic NiVER well, and that our implementation techniques runs faster. The last column shows SatELite with all reductions on, which results in a strict improvement in size.

generated from the *PicoJava* design[7] and 13 liveness problems from *SatLib*.[8] The result contains 115 CNF files.

***Study 1 – Comparing reduction rates with* NiVER.** This study shows that SatELite is an improvement over earlier work. We use the same problem set as presented in the NiVER paper [21]. The results are shown in *Table 1*.

Study 2 – Reduction rates and preprocessing time. *Figure 1* shows the effect of applying preprocessing in terms of the number of remaining variables, clauses, and literal occurrences. We see that for most problems the number of clauses drop significantly, as well as the number of literal occurrences (with some exceptions), resulting in smaller CNFs and faster unit propagation.

The runtime of the preprocessing is also plotted in relation to the time of solving the original CNF. For problems requiring between 30 seconds and 30 minutes to solve, preprocessing took less than 1/10th of the total time. Only for some of the easiest problems did preprocessing dominate runtime, but never in any really harmful way.

Study 3 – Runtime comparison solving with/without preprocessing. In *Figure 2* we plotted the preprocessing plus SAT solving time using the strongest version of our preprocessing (y-axis) against SAT solving without preprocessing (x-axis). Although not a consistent improvement time-wise, in the big majority of cases preprocessing lead to a significant performance increase. In particular for ZChaff, the improvement was virtually exceptionless.

[7] www.sun.com/microelectronics/communitysource/picojava/download.html
[8] www.cs.ubc.ca/ hoos/SATLIB/Benchmarks/SAT/BMC/bmc.tar.gz

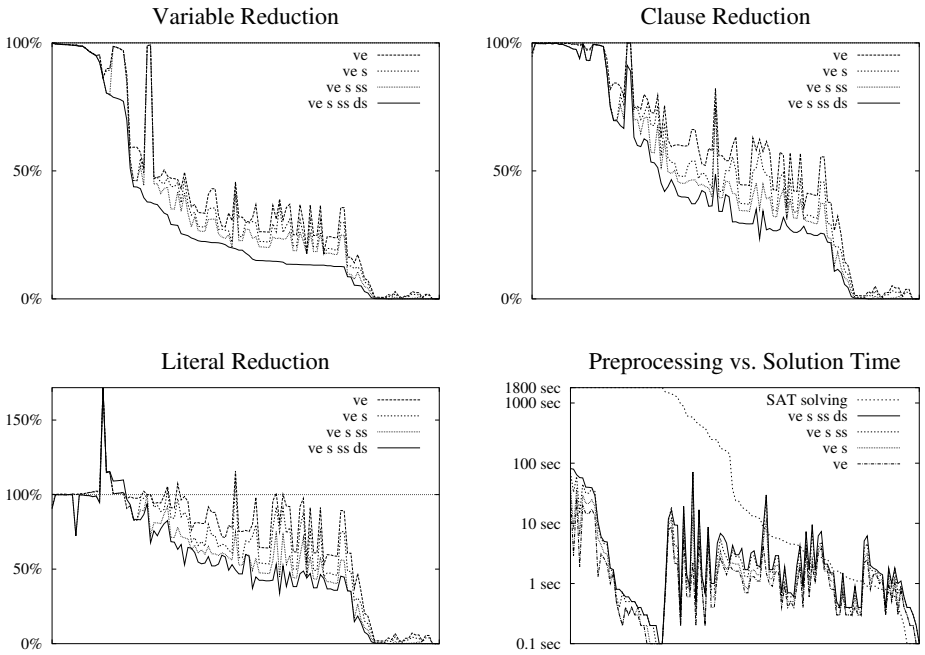

Fig. 1. *Relative Reduction and Preprocessing Time.* **The plots show the remaining size of *each one* of the 115 problems in the Industrial Mix** after reduction by our preprocessing techniques. The abbreviations are: "ve" variable elimination, "s" subsumption, "ss" self-subsumption, "ds" definitional substitution. The reduction is measured relative to the size of the original CNF after applying unit propagation (= 100%). The curves show the remaining variables (upper left), clauses (upper right), and literals (lower left). For all three plots the instances on the x-axis are sorted in the same way. **The order is determined by the percentage of remaining variables for the most *effective* version of the preprocessor** (the lower curve in the upper left plot labelled "ve s ss ds"). Our primary simplification target, the elimination of variables induces a simplification of the number of clauses in most cases as well. The number of literals follows more loosely the same trend. These three plots also show that our new simplification techniques are very effective compared to the approach taken by NiVER [21], which corresponds to the curves labelled "ve". Often an additional factor of two in reduction can be achieved.

The lower right plot shows in logarithmic scale the absolute time needed for preprocessing in relation to the overall solution time. The upper curve refers to the time for solving an instance with MiniSat not using preprocessing (timeout set to 1800 seconds). The remaining five curves show only the time used for preprocessing alone with decreasing effort. Preprocessing time turns out to be negligible compared to the overall solution time in most cases, even when our most aggressive techniques are used. Only for very simple instances is it better to run the solver without preprocessing.

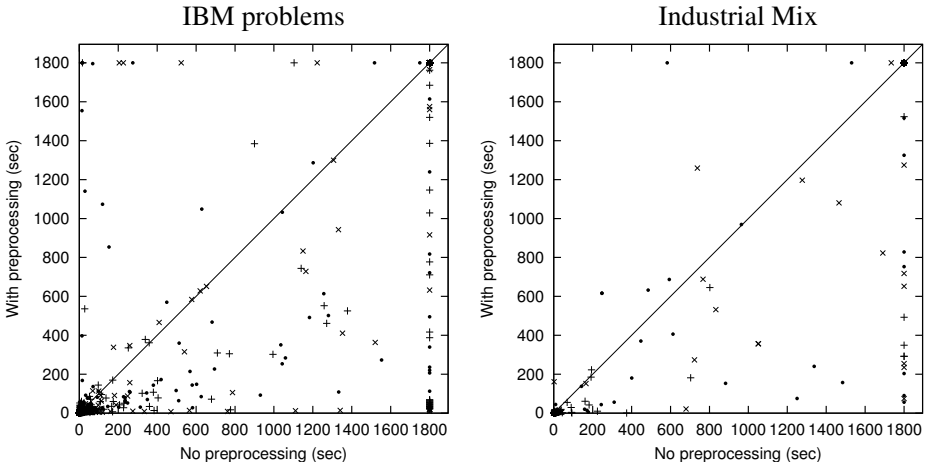

Fig. 2. *Heads-up comparison, with and without preprocessing.* The graphs shows a time comparison between "SAT solving" and "preprocessing + SAT solving" (using all reduction techniques). A mark below the diagonal means faster total solving time when first applying preprocessing. A dot (•) represents MiniSat, a plus (+) ZChaff, and a cross (×) BerkMin. A timeout of 1800 seconds was used, and marks along the edges represent tests which timed out for one of the two executions.

Study 4 – Runtime effect of the different techniques. To evaluate the benefit of the different levels of reduction, we run all three SAT solvers on all benchmarks, both the IBM Problems and the Industrial Mix, with 5 different levels of optimization: (1) Nothing (original CNF after propagating unit clauses), (2) only variable elimination, (3) variable elimination plus subsumption, (4) variable elimination, subsumption and self-subsumption, (5) variable elimination using definitional substitution (when possible), subsumption and self-subsumption.

The result is plotted in *Figure 5*. The curves show that not only are more problems solved fast by preprocessing, but also more problems in total when a long timeout is given.

Study 5 – Incremental k-induction. In *Table 2* and *Figure 4*, a small study of applying our reduction techniques in an incremental context is presented. The internal SAT solver of SatElite, a less optimized version of MiniSat, allows SatElite to be used not only as a preprocessor, but also as an incremental SAT solver. Simplification is applied between each incremental SAT problem. Although this is a small study, the preliminary results suggest that our techniques pay off in an incremental context too.

6 Conclusion

New simplification techniques were presented together with important implementation details. On a large representative set of industrial benchmarks it was shown, that they speed up SAT solvers considerably. We also believe that preprocessing techniques partially provide a solution to the important problem of

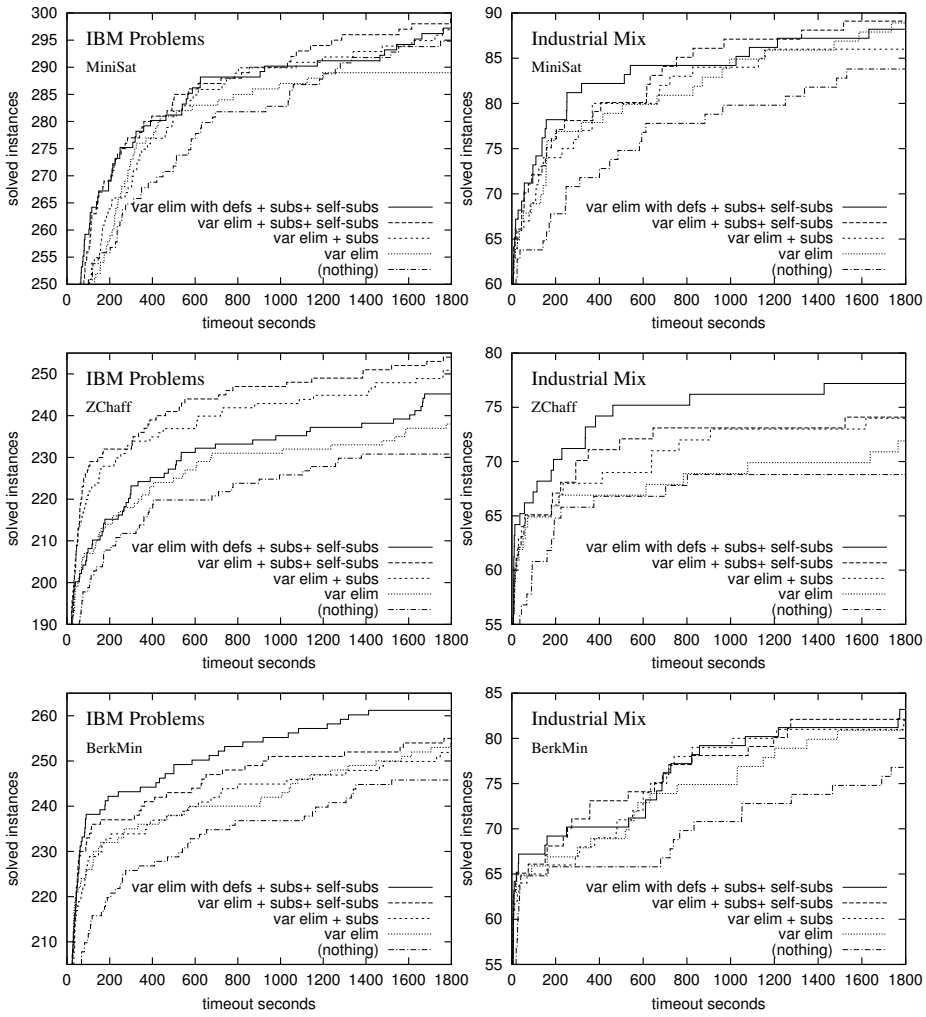

Fig. 3. *Comparing different preprocessing options using* SatELite. Time includes both preprocessing and solving. Even though variable elimination by definitional substitution gives a consistent reduction compared to variable elimination by clause distribution, it is not a clear winner in terms of CPU time, but seems to depend on which solver you apply (the solid line vs. the long-dashed line). However, both lines are clearly above the "(nothing)"-line, representing no preprocessing. The addition of self-subsumption to normal subsumption seems to be a clear winner (often better, never worse). To get an estimate of the speedup, the graphs could be read by fixing a particular number of solved instances, and see what timeout is required to solve that number of instances. On the IBM benchmarks, MiniSat requires a timeout of about 250 seconds to solve 275 problems with full preprocessing, but a timeout of more than 600 seconds with no preprocessing.

Name	Depth	Plain		Simplifying	
vis.prodcell_12	29	62.7 s	(266k)	25.3 s	(88k)
vis.prodcell_14	16	7.8 s	(124k)	6.4 s	(29k)
vis.prodcell_15	23	30.5 s	(200k)	14.0 s	(56k)
vis.prodcell_17	27	64.5 s	(253k)	24.1 s	(76k)
vis.prodcell_18	13	7.5 s	(114k)	5.0 s	(35k)
vis.prodcell_19	22	19.2 s	(192k)	12.3 s	(55k)
vis.prodcell_23	13	9.5 s	(120k)	5.9 s	(37k)
vis.prodcell_24	37	120.5 s	(319k)	34.3 s	(94k)

Table 2. *Study on k-induction.* We modified TIP [9] to use SATELITE as a backend and ran the *zigzag* incremental induction algorithm on the "prodcell" problem distributed with VIS. The table shows the total runtime of each problem in seconds, omitting examples solved in less than 1 second. Within parenthesis, the number of clauses of the final incremental SAT instance is printed. In the rightmost column, all simplifications of SATELITE were invoked between each incremental step. The "depth" is the induction depth needed to prove the property (all properties are true).

generating good CNFs in the application domain of circuit verification. As future work, it would be interesting to compare SAT solving time on problems that have been (1) clausified in a good way, and (2) clausified in a naive way, but processed with SATELITE. We also want to combine and compare our preprocessing techniques with the orthogonal techniques mentioned in the introduction.

Acknowledgements

Niklas Eén wants to thank Niklas Sörensson for setting him of in the direction of using self-subsumption in SAT.

References

1. F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Proc. SAT'03*, volume 2919 of *LNCS*.
2. A. Biere. Resolve and expand. In *Prel. Proc. SAT'04*.
3. T. Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14, 1992.
4. R. Brafman. A simplifier for propositional formulas with many binary clauses. *IEEE Trans. on Systems, Man, and Cybernetics*, 34(v1), 2004.
5. P. Chatalic and L. Simon. ZRes: The old Davis-Putnam procedure meets ZBDDs. In *Proc. CADE'00*, number 1831 in LNAI.
6. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Comm. of the ACM*, 5(7), 1962.
7. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3), 1960.
8. N. Eén and N. Sörensson. An extensible SAT solver. In *Proc. SAT'03*, volume 2919 of *LNCS*.
9. N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. In *Proc. BMC'03*, volume 89(4) of *ENTCS*.

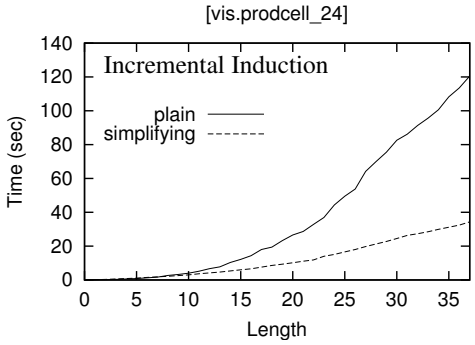

Fig. 4. *Case study on incremental k-induction.* The curve shows the progress of the temporal induction algorithm of TIP [9] running on the hardest example of *Table 2*. The graph plots the total execution time (y-axis) of all induction steps performed upto a certain length (x-axis). In this particular experiment, each step after length 25 takes more or less constant time.

10. E. Giunchiglia, M. Maratea, and A. Tacchella. Dependent and independent variables for propositional satisfiability. In *Proc. JELIA'02*, volume 2424 of LNCS.
11. É. Grégoire, R. Ostrowski, B. Mazure, and L. Saïs. Automatic extraction of functional dependencies. In *Prel. Proc. SAT'04*.
12. P. Jackson and D. Sheridan. Clause form conversions for boolean circuits. In *Prel. Proc. SAT'04*.
13. A. Kühlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 21(12), 2002.
14. W. Kunz and D. Pradhan. Recursive learning: An attractive alternative to the decision tree for test generation in digital circuits. In *Proc. ITC'92*.
15. I. Lynce and J. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *Proc. ICTAI'03*.
16. Y. Novikov. Local search for boolean relations on the basis of unit propagation. In *Proc. of DATE'03*.
17. R. Ostrowski, É. Grégoire, B. Mazure, and L. Saïs. Recovering and exploiting structural knowledge from CNF formulas. In *Proc. CP'02*, volume 2470 of *LNCS*.
18. D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3), 1986.
19. N. Sörensson. Conflict clause simplification using subsumption resolution. paper in preparation.
20. G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula, 1989. Swedish Patent N° 467 076.
21. S. Subbarayan and D. Pradhan. NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In *Prel. Proc. SAT'04*.
22. G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constr. Math. and Math. Logic*, 1968.
23. M. Velev. Efficient translation of boolean formulas to CNF in in formal verification of microprocessors. In *Proc. ASP-DAC'04*.