# Preprocessing in Incremental SAT

Alexander Nadel [1] Vadim Rivchyn[1,2] Ofer Strichman[2]
alexander.nadel@intel.com rvadim@tx.technion.ac.il
ofers@ie.technion.ac.il

[1] Design Technology Solutions Group, Intel Corporation, Haifa, Israel
[2] Information Systems Engineering, IE, Technion, Haifa, Israel

**Abstract.** Preprocessing of CNF formulas is an invaluable technique when attempting to solve large formulas, such as those that model industrial verification problems. Unfortunately, the best combination of preprocessing techniques, which involve variable elimination combined with subsumption, is incompatible with incremental satisfiability. The reason is that soundness is lost if a variable is eliminated and later reintroduced. *Look-ahead* is a known technique to solve this problem, which simply blocks elimination of variables that are expected to be part of future instances. The problem with this technique is that it relies on knowing the future instances, which is impossible in several prominent domains. We show a technique for this realm, which is empirically far better than the known alternatives: running without preprocessing at all or applying preprocessing separately at each step.

## 1 Introduction

Whereas CNF preprocessing techniques have been known for a long time (e.g., [1, 2]), most are not cost-effective when it comes to formulas with millions of clauses – a typical size for industrial verification problems that are being routinely solved these days in the EDA industry. In that respect one of the major breakthroughs in practical SAT solving in the last few years has been the combined preprocessing techniques that were suggested by Een and Biere [3]: non-increasing variable elimination through resolution, coupled with subsumption and self-subsumption. These three techniques remove variables, clauses and literals, respectively. They are implemented in MiniSat [4] and the stand-alone preprocessor SatELite, and are in common use by many SAT solvers. Our experience with industrial verification instances shows that these techniques frequently remove more than half of the formula, and enable the solving of large instances that otherwise cannot be solved within a reasonable time limit. We will describe these techniques in more detail in Sect. 2.

A known problem with variable elimination is the fact that it is incompatible, at least in its basic form as published, with incremental SAT solving [4, 9, 10]. The reason, as was pointed out already in [3], is that variables that are eliminated may reappear in future instances. Soundness is not maintained in this scenario. For example, suppose that a formula contains the two clauses $(a \vee v), (b \vee \bar{v})$.

Eliminating $v$ results in removing these two clauses and adding the resolvent $(a \vee b)$. Suppose, now, that in the next instance the clauses $(\bar{a}), (\bar{v})$ are added, which clearly contradict $(a \vee v)$. Yet since we erased that clause and since there is no contradiction between the resolvent and the new clauses, the new formula is possibly satisfiable — soundness is lost.

A possible remedy to this problem which was already suggested in [3] and experimented with in [7], is *look-ahead*. This means that variables that are known to be added in future instances are not eliminated. The problem with look-ahead is that it is not always possible, because information about future instances is not always available. Examples of such problem domains are:

- Some applications require interactive communication with the user for determining the next portion of the problem. For example, a recent article from IBM [3] describes a process in which the verification engineer may re-invoke the same instance of the SAT-based model checker for verifying a new property, which is not known a-priory (it depends on the result of the previous property). In such a case only a small part of the formula is changed, and hence incremental satisfiability may be crucial for performance.
- In some applications the calculation of the next portion of the problem depends on the results of the previous invocation of the SAT solver. For example, various tasks in MicroCode validation [6] are solved by using a symbolic execution engine to explore the paths of the program. The generated proof obligations are solved by an incremental SAT-based SMT solver. In this application, the next explored path of the program is determined based on the result of the previous computation.
- In Intel, the conversion of BMC problems to CNF is done after applying a 'saturation' optimization at the circuit level. Saturation divides all the variables into equivalence classes and tries to unite them by propagating short clauses that were learned in a previous instance — hence the dependency that prevents precalculating the instances. The SAT solver is provided only with the representatives of the equivalence classes. As a result, simple unrolling cannot predict those variables that will be present or absent in future instances.

Another possible remedy is called *full preprocessing*. It was briefly mentioned in [7] as an option that is expected not to scale, although in our experiments it is occasionally competitive. The idea is to perform full preprocessing before each instance. This means that all variables that were previously eliminated are returned to the formula and resolvents are removed, other than those that subsumed other clauses and hence cannot be removed. Therefor preprocessing is performed independently of past or future instances, other than the fact that it marks subsuming resolvents. The disadvantage of this approach comparing to *incremental preprocessing* — the main contribution of this article — is that it repeats a lot of work that has already been done in previous instances. Our experiments with large instances show that this extra overhead can add more than an hour to the preprocessing time.

In this article we suggest a method for combining the method of [3] with assumptions-based incremental SAT [4]. Our experiments show that it is much better than either running without preprocessing at all or full preprocessing. Look-ahead is still better overall, however, when possible. The solution we suggest is simple and rather easy to implement. Basically we eliminate variables regardless of future instances, and every time a variable is reintroduced into the formula we choose whether to *reeliminate*, or *reintroduce* it. An exception is made for the assumptions variables, which must be reintroduced. For both routes we need to save the clauses that were erased in the process of elimination: these need to be resolved with the new clauses for the former, and returned to the formula for the latter. As we show, the *order* in which variables are reeliminated or reintroduced matters for correctness. Specifically, the order must be *consistent* between instances. The order also changes the resulting reduced formula and hence the solving time. Our experiments show that in most cases the consistent order reduces the solving time.

We continue in the next section by describing the technical details of variable elimination, subsumption and self-subsumption. In Sect. 3 we present incremental preprocessing, which is an adaptation of these algorithms to the setting of incremental SAT. In Sect. 4 we summarize the results of our extensive experiments with industrial verification benchmarks from Intel.

## 2 Preliminaries

Let $\varphi$ be a CNF formula. We denote by $vars(\varphi)$ the variables used in $\varphi$. For a clause $c$ we write $c \in \varphi$ to denote that $c$ is a clause in $\varphi$. For $v \in vars(\varphi)$ we define $\varphi_v = \{c \mid c \in \varphi \wedge v \in c\}$ and $\varphi_{\bar{v}} = \{c \mid c \in \varphi \wedge \bar{v} \in c\}$ (somewhat abusing notation, as we refer here to $v$ as both a variable and a literal). Our setting includes the use of *assumptions* [5].

**Variable elimination**
Input: formula $\varphi$ and a variable $v \in vars(\varphi)$.
Output: formula $\varphi'$ such that $v \notin vars(\varphi')$ and $\varphi'$ and $\varphi$ are equisatisfiable.

Typically this preprocessing is applied only if the number of clauses in $\varphi'$ is not larger than in $\varphi$. More generally one may define a positive limit on the growth in the number of clauses, but for simplicity we will assume here that this limit is 0. Alg. 1 presents a variable elimination algorithm, where the eliminated variable $v$ is the parameter. The variable $v$ must be unassigned.

The function Resolve computes the set of non-tautological resolvents of two sets of clauses given to it as input (the check in line 5 excludes tautological resolvents). Function EliminateVar uses Resolve to compute the set $Res$ of such resolvents of $\varphi_v$ and $\varphi_{\bar{v}}$. If this set is larger than $|\varphi_v|+|\varphi_{\bar{v}}|$ it simply returns, and hence $v$ is not eliminated. Otherwise in line 4 it adds the resolvents $Res$ and discards the resolved clauses. All the variables in the resolvents are added to a list $TouchedVars$ in line 6. This list will be used later, in Alg. 2, for driving further subsumption and self-subsumption.

**Algorithm 1** A variable elimination algorithm similar to the one implemented in MiniSat and in [3].

```
1: function Resolve(clauseset pos, clauseset neg)
2:     clauseset res = ∅;
3:     for each clause p ∈ pos do
4:         for each clause n ∈ neg do
5:             if p and n have a single possible pivot then
6:                 res = res ∪ resolution(p, n);
7:     return res;

1: function EliminateVar(var v)
2:     clauseset Res = Resolve (φ_v, φ_v̄);
3:     if |Res| > |φ_v| + |φ_v̄| then return ∅;          ▷ no variable elimination
4:     φ = (φ ∪ Res) \ (φ_v ∪ φ_v̄);
5:     ClearDataStructures(v);        ▷ clearing occurrence list, watch-list, scores-list
6:     TouchedVars = TouchedVars ∪ vars(Res);                    ▷ used in Alg. 2
7:     return Res;
```

**Subsumption**
*Input*: $\varphi \wedge (l_1 \vee \cdots \vee l_i) \wedge (l_1 \vee \cdots \vee l_i \vee l_{i+1} \vee \cdots \vee l_j)$.
*Output*: $\varphi \wedge (l_1 \vee \cdots \vee l_i)$.

**Self-subsumption**
*Input*: $\varphi \wedge (l_1 \vee \cdots \vee l_i \vee l) \wedge (l_1 \vee \cdots \vee l_i \vee l_{i+1} \vee \cdots \vee l_j \vee \bar{l})$.
*Output*: $\varphi \wedge (l_1 \vee \cdots \vee l_i \vee l) \wedge (l_1 \vee \cdots \vee l_i \vee l_{i+1} \vee \cdots \vee l_j)$.

**Preprocessing** The preprocessing algorithm described in Alg. 2 is similar to that implemented in MiniSat 2.2 [4] (based on the stand-alone preprocessor SatELite [3]). *SubsumptionQ* is a global queue of clauses. For each $c \in$ *SubsumptionQ*, and each $c' \in \varphi$, RemoveSubsumptions (1) checks if $c \subset c'$ and if yes performs subsumption, and otherwise (2) if $c$ self-subsumes $c'$ then it performs self-subsumption. Essentially it is similar to the implementation suggested in [3]. Self-subsumption is followed by adding the reduced clause back to the queue. The function runs until the queue is empty. Note that assumptions are not eliminated. Eliminating assumptions would render the algorithm unsound.

In line 5 the variables are scanned in an increasing order of occurrences count. Note that in line 7 RemoveSubsumptions is applied only to the set of newly generated resolvents.

## 3 Incremental preprocessing

We now describe an incremental version of the preprocessing algorithm. In contrast to the full-preprocessing algorithm that was briefly described in the introduction (performing preprocessing of the new formula, together with learned

**Algorithm 2** Preprocessing, similar to the algorithm implemented in MiniSat 2.2.

```
1: function PREPROCESS
2:     SubsumptionQ = φ;
3:     while SubsumptionQ ≠ ∅ do
4:         REMOVESUBSUMPTIONS ();
5:         for each unassigned non-assumption variable v do      ▷ order heuristically
6:             SubsumptionQ = ELIMINATEVAR (v);
7:             if SubsumptionQ ≠ ∅ then REMOVESUBSUMPTIONS ();
8:         SubsumptionQ = {c | vars(c) ∩ TouchedVars ≠ ∅};
9:         TouchedVars = ∅;
```

clauses from previous instances), our suggested algorithm does not repeat preprocessing work that was done in previous instances.

In our setting of incremental SAT, each instance is given as a set of clauses that should be added to the formula accumulated thus far. Removal of clauses is done indirectly, by using assumptions that are clause selectors. For example, if $v$ is an assumption variable, then we can add $\bar{v}$ to a set of clauses. Assigning this variable FALSE is equivalent to removing this set.

Let $\varphi^0$ denote the initial formula, and $\Delta^i$ denote the set of clauses added at step $i$. Step $i$ for $i > 0$ begins with a formula denoted $\varphi^i$, initially assigned the conjunction of $\varphi^{i-1}$ at the *end* of the solving process (i.e., after being preprocessed and with additional learned clauses), and $\Delta^i$. This formula changes during the solving process.

Preprocessing in an incremental SAT setting requires various changes. In step $i$, the easy case is when we wish to eliminate a variable $v$ that is *not* eliminated in step $i-1$. ELIMINATEVAR-INC, shown in Alg. 3 is a slight variation of ELIMINATEVAR that we saw in Alg. 1. The only difference is that if $v$ is eliminated, then it saves additional data that will be used later on, as we will soon see. Specifically, it saves $\varphi^i_v$ and $\varphi^i_{\bar{v}}$ in clause-sets denoted respectively by $S_v$ and $S_{\bar{v}}$, and in the next line also the number of resolvents in a queue called $ElimVarQ$. This queue holds tuples of the form $\langle$variable $v$, int $resolvents\rangle$.

**Algorithm 3** Variable elimination for $\varphi^i$, where the eliminated variable $v$ was *not* eliminated in $\varphi^{i-1}$.

```
1: function ELIMINATEVAR-INC(var v, int i)
2:     clauseset Res = RESOLVE (φ^i_v, φ^i_v̄);
3:     if |Res| > |φ^i_v| + |φ^i_v̄| then return ∅;          ▷ no variable elimination
4:     S_v = φ^i_v; S_v̄ = φ^i_v̄;                          ▷ Save for possible reintroduction
5:     ElimVarQ.push(⟨v, |Res|⟩);                          ▷ Save #resolvents in queue
6:     φ^i = (φ^i ∪ Res) \ (φ^i_v ∪ φ^i_v̄);
7:     CLEARDATASTRUCTURES (v);
8:     TouchedVars = TouchedVars ∪ vars(Res);              ▷ used in Alg. 5
9:     return Res;
```

The more difficult case is when $v$ is already eliminated at step $i-1$. In that case we invoke ReEliminate-Or-ReIntroduce, as shown in Alg. 4. This function decides between reintroduction and reelimination.

- *Reelimination.* In Line 6 the algorithm computes the set of resolvents $Res$ that need to be added in case $v$ is reeliminated. Note that $\varphi^i$ may contain $v$ because of two separate reasons. First, $vars(\Delta^i)$ may contain $v$; Second, variables that were reintroduced in step $i$ *prior* to $v$ may have led to reintroduction of clauses that contain $v$. The total number of resolvents resulting from eliminating $v$ is $|Res|$ + the number of resolvents incurred by eliminating $v$ up to step $i$, which, recall, is saved in $ElimVarQ$.
- *Reintroduction.* In case we decide to cancel elimination, the previously removed clauses $S_v$ and $S_{\bar{v}}$ have to be reintroduced. The total number of clauses resulting from reintroducing $v$ is thus $|S_v \cup S_{\bar{v}} \cup \varphi^i_v \cup \varphi^i_{\bar{v}}|$. Note that the algorithm reintroduces variables that appear in the assumption list.

The decision between the two options is made in line 7. If reintroduction results in a smaller number of clauses, we simply return the saved clauses $S_v$ and $S_{\bar{v}}$ by calling ReIntroduceVar, which also removes its entry from $ElimVarQ$ because $v$ is no longer eliminated. The rest of the code is self-explanatory.

**Algorithm 4** Variable elimination for $\varphi^i$, where the eliminated variable (located in $ElimVarQ[loc].v$) was already eliminated in $\varphi^{i-1}$.

```
1: function ReIntroduceVar(var v, int loc, int i)
2:     φ^i += S_v ∪ S_v̄;
3:     erase ElimVarQ[loc];                    ▷ v is not eliminated, hence 0 resolvents

1: function ReEliminateVar(clauseset Res, var v, int loc, int i)
2:     S_v = S_v ∪ φ^i_v; S_v̄ = S_v̄ ∪ φ^i_v̄;
3:     ElimVarQ[loc].resolvents += |Res|;
4:     φ^i = (φ^i ∪ Res) \ (φ^i_v ∪ φ^i_v̄);
5:     ClearDataStructures (v);
6:     TouchedVars = TouchedVars ∪ vars(Res);

1: function ReEliminate-Or-ReIntroduce(int loc, int i)
2:     var v = ElimVarQ[loc].v;                ▷ The variable to eliminate
3:     if v is an assumption then
4:         ReIntroduceVar(v, loc, i);
5:         return ∅;
6:     clausesetRes = Resolve(φ^i_v, φ^i_v̄) ∪
                      Resolve(φ^i_v, S_v̄) ∪ Resolve(S_v, φ^i_v̄);
7:     if (|Res| + ElimVarQ[loc].resolvents) > |S_v ∪ S_v̄ ∪ φ^i_v ∪ φ^i_v̄| then
8:         ReIntroduceVar(v, loc, i);
9:         return ∅;
10:    ReEliminateVar (Res, v, loc, i);
11:    return Res
```

Given EliminateVar-Inc and ReEliminate-Or-ReIntroduce we can now focus on Preprocess-inc in Alg. 5, which is parameterized by the instance number $i$. The difference from Alg. 2 is twofold: First, variables that are already eliminated in the end of step $i-1$ are processed by ReEliminate-Or-ReIntroduce; Second, other variables are processed in EliminateVar-inc. The crucial point here is the *order* in which variables are eliminated. Note that 1) elimination is consistent between instances, and 2) variables that are not currently eliminated are checked for elimination only at the end. These two conditions are necessary for correctness, because, recall, ReIntroduceVar may return clauses that were previously erased. These clauses may contain any variable that was not eliminated at the time they were erased.

*Example 1.* Suppose that in step $i-1$, $v_1$ was eliminated, and as a result a clause $c = (v_1 \vee v_2)$ was removed. Then $v_2$ was eliminated as well. Suppose now that in step $i$ we first reeliminate $v_2$, and then decide to reintroduce $v_1$. The clause $c$ above is added back to the formula. But $c$ contains $v_2$ which was already eliminated. □

**Algorithm 5** Preprocessing in an incremental SAT setting

```
1: function Preprocess-inc(int i)                          ▷ preprocessing of φ^i
2:     SubsumptionQ = {c | ∃v. v ∈ c ∧ v ∈ vars(Δ^i)};
3:     while SubsumptionQ ≠ ∅ do
4:         RemoveSubsumptions ();
5:         for (j = 0 ... |ElimVarQ| − 1) do      ▷ scanning eliminated vars in order
6:             v = ElimVarQ[j].v;
7:             if |φ^i_v| = |φ^i_v̄| = 0 then continue;
8:             ReEliminate-Or-ReIntroduce (j, i);
9:         for each unchecked non-assumption variable v do      ▷ scanning the rest
10:            SubsumptionQ = EliminateVar-inc (v, i);
11:            RemoveSubsumptions ();
12:        SubsumptionQ = {c | vars(c) ∩ TouchedVars ≠ ∅};
13:        TouchedVars = ∅;
```

Let $\psi^n = \varphi^0 \wedge \bigwedge_{i=1}^{n} \Delta^i$, i.e., $\psi_n$ is the $n$-th formula without preprocessing at all. We claim that:

**Proposition 1.** *Algorithm* Preprocess-inc *is correct, i.e., for all* $n$

$$\psi^n \text{ is equisatisfiable with } \varphi^n \ .$$

*Proof.* The full proof is given in a technical report [8]. Here we only sketch its main steps. The proof is by induction on $n$. The base case corresponds to standard (i.e., non-incremental) preprocessing. Proving the step of the induction relies on another induction, which proves that the following two implications hold

right after line 8 at the $j$-th iteration of the first loop in PREPROCESS-INC, for $j \in [0 \dots |ElimVarQ| - 1]$:

$$\psi^n \implies \Big(\varphi^n \wedge \bigwedge_{k=j+1}^{|ElimVarQ|-1} \bigwedge_{c \in S_{v_k} \cup S_{\bar{v}_k}} c\Big) \implies \exists v_1 \dots v_j.\ \psi^n \ ,$$

The implication on the right requires some attention: existential quantification is necessary because of variable elimination via resolution (in the same way that $Res(x \vee A)(\bar{x} \vee B) = (A \vee B)$ and $(A \vee B) \implies \exists x.\ (x \vee A)(\bar{x} \vee B)$). The crucial point in the proof of this implication is to show that if a variable is eliminated at step $j$, it cannot reaapear in the formula in later iterations. This is indeed guaranteed by the order in which the first loop processes the variables.

Note that at the last iteration $j = |ElimVarQ| - 1$ and the big conjunctions disappear. This leaves us with

$$\psi^n \implies \varphi^n \implies \exists v_1 \dots v_j.\ \psi^n \ ,$$

which implies that $\psi^n$ is equisatisfiable with the formula after the last iteration. The second loop of PREPROCESS-INC is non-incremental preprocessing, and hence clearly maintains satisfiability. □

**Removal of resolvents** Recall that REINTRODUCEVAR returns the clause sets $S_v$ and $S_{\bar{v}}$ to the formula. So far we ignored the question of what to do with the resolvents: should we remove them given that we canceled the elimination of $v$? These clauses are implied by the original formula, so keeping them does not hinder correctness. *Removing* them, however, is not so simple, because they may have participated in subsumption / self-subsumption of other clauses. Removing them hinders soundness, as demonstrated by the following example.

*Example 2.* Consider the following four clauses:

$$c_1 = (l_1 \vee l_2 \vee l_3) \quad c_2 = (l_4 \vee l_5 \vee \bar{l_3})$$
$$c_3 = (l_1 \vee l_2 \vee \bar{l_4}) \quad c_4 = (l_1 \vee l_2 \vee \bar{l_5}) \ ,$$

and the following sequence:

- elimination of $var(l_3)$:
  - $c_5 = res(c_1, c_2) = (l_1 \vee l_2 \vee l_4 \vee l_5)$ is added;
  - $c_1$ and $c_2$ are removed and saved.
- self-subsumption between $c_3$ and $c_5$: $c_5 = (l_1 \vee l_2 \vee l_5)$.
- self-subsumption between $c_4$ and $c_5$: $c_5 = (l_1 \vee l_2)$.
- subsumption of $c_3$ and $c_4$ by $c_5$.
- removal of the resolvent $c_5$ and returning of $c_1$ and $c_2$.

We are left with only a subset of the original clauses ($c_1$ and $c_2$), which do not imply the rest. In this case the original formula is satisfiable, but it is not hard to see that the subsumed clauses $(c_3, c_4)$ could have been part of an unsatisfiable set of clauses, and hence that their removal could have changed the result from unsat to sat. Soundness is therefore not secured if resolvents that participated in subsumption are removed. □

In our implementation we solve this problem as follows. When eliminating $v$, we associate all the resolvent clauses with $v$. In addition, we mark all clauses that subsumed other clauses. We then change ReIntroduceVar as can be seen in Alg. 6. Note that in line 3 we guarantee that unit resolvents remain: it does not affect correctness and is likely to improve performance.

**Algorithm 6** ReIntroduceVar with removal of resolvents that did not participate in subsumption.

```
1: function ReIntroduceVar(var v, int loc, int i)
2:     φ^i += S_v ∪ S_v̄;
3:     for each non-unit clause c associated with v do
4:         if c is not marked then Remove c from φ^i;
5:     erase ElimVarQ[loc];
```

## 4 Experimental results

We implemented incremental preprocessing on top of Fiver[1], and experimented with hundreds of large processor Bounded Model-checking instances from Intel, categorized to four different families. In each case the problem is defined as performing BMC up to a given bound[2] in increments of size 1, or finding a satisfying assignment on the way to that bound. The time out was set to 4000 sec. After removing those benchmarks that cannot be solved by any of the tested methods within the time limit we were left with 206 BMC problems.[3] We turned off the 'saturation' optimization at the circuit level that was described in the introduction, in order to be able to compare our results to look-ahead. Overall in about half of the cases there is no satisfying assignment up to the given bound.

The first graph, in Fig. 1, summarizes the overall results of the four compared methods: full-preprocessing, no-preprocessing, incremental-preprocessing, and look-ahead. The number of time-outs and the average total run-time with these four methods is summarized in Table 1.

Look-ahead wins overall, but recall that in this article we focus on scenarios in which lookahead is impossible. Also note that it only has an advantage in a setting in which there is a short time-out. Incremental-preprocessing is able to close the gap and become almost equivalent once the time-out is set to a high value. It seems that the reason for the advantage of incremental preprocessing over look-ahead in hard instances is that unlike the latter, it does not force each variable to stay in the formula until it is known that it will not be added from thereon.

[1] Fiver is a new SAT solver that was developed in Intel. It is a CDCL solver, combining techniques from Eureka, Minisat, and other modern solvers.

[2] Internal customers in Intel are typically interested in checking properties up to a given bound.

[3] The benchmarks are available upon request from the authors.

| Method | Time-outs | Avg. total run-time |
|---|---|---|
| full-preprocessing | 68 | 2465.5 |
| no-preprocessing | 42 | 1784.7 |
| incremental-preprocessing | 2 | 1221.3 |
| look-ahead | 0 | 1064.9 |

**Table 1.** The number of time-outs and the average total run time (incl. preprocessing) achieved by the four compared methods.

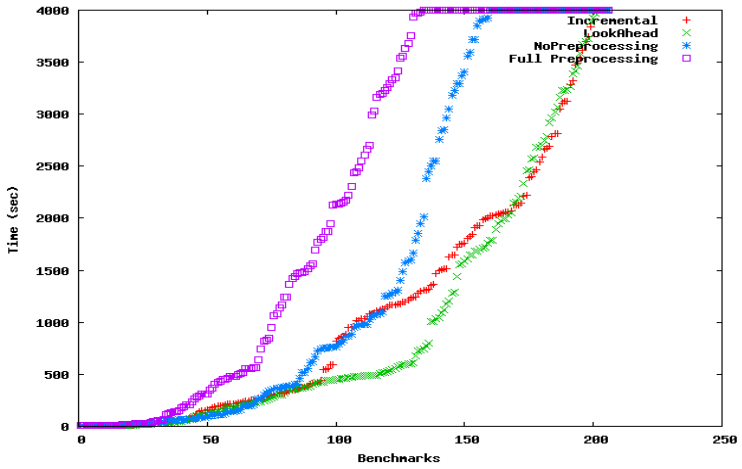


**Fig. 1.** Overall run-time of the four compared methods.

We now examine the results in more detail. Fig. 2 shows the consistent benefit of incremental preprocessing over full preprocessing. The generated formula is not necessarily the same because of the order in which the variables are examined. Recall that it is consistent between instances in Preprocess-Inc and gives priority to those variables that are currently eliminated. In full preprocessing, on the other hand, it checks each time the variable that is contained in the minimal number of clauses. The impact of the preprocessing order on the search time is inconsistent, but there is a slight advantage to that of Preprocess-Inc, as can be seen in the middle figure. The overall run time favors Preprocess-Inc, as can be seen at the bottom figure.

Fig. 3 compares incremental preprocessing and no preprocessing at all. Again, the advantage of the former is very clear.

Finally, Fig. 4 compares incremental preprocessing and look-ahead, which shows the benefit of knowing the future. The fact that the preprocessing time of the latter is smaller is very much expected, because it does not have the overhead incurred by the checks in Alg. 3 and the multiple times that each variable can be reeliminated and reintroduced. The last graph shows that a few more instances

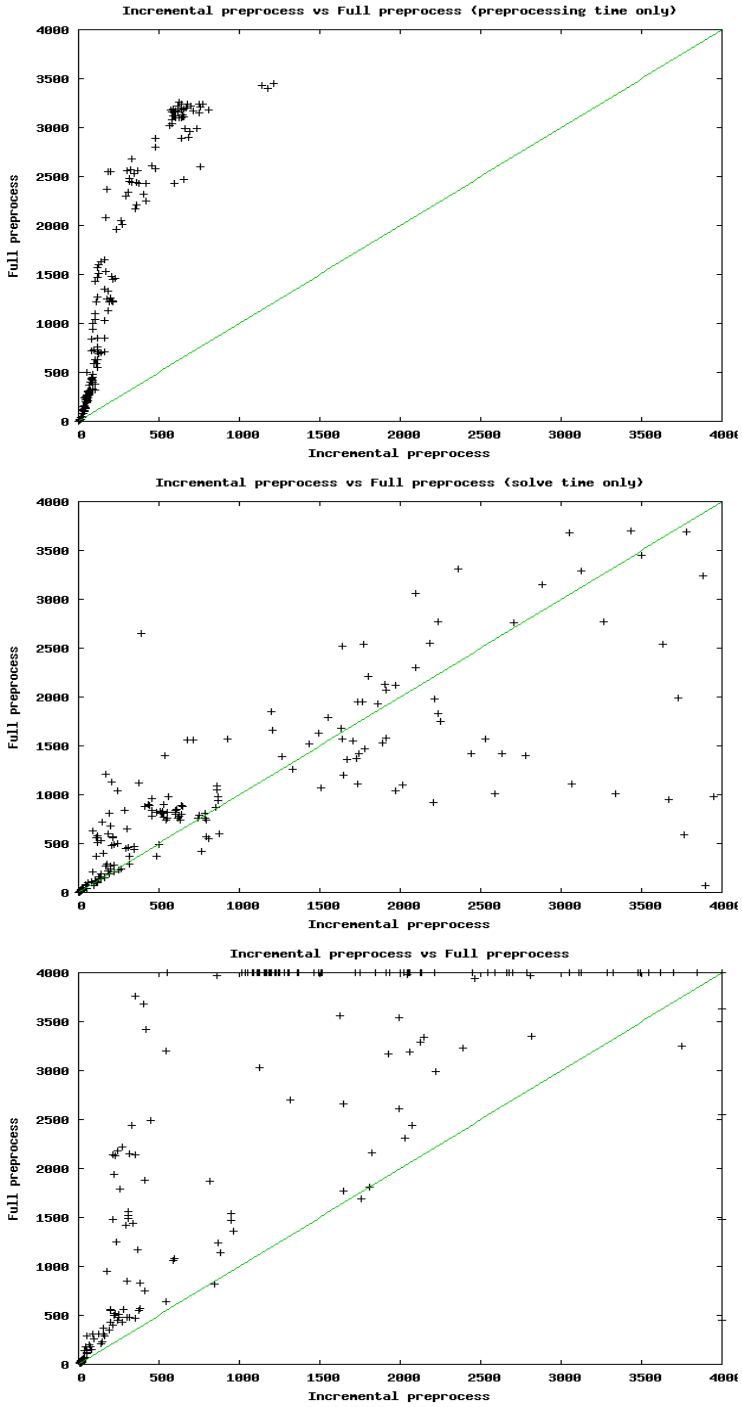


**Fig. 2.** Incremental preprocessing vs. full preprocessing: (top) preprocessing time, (middle) SAT time, and (bottom) total time.

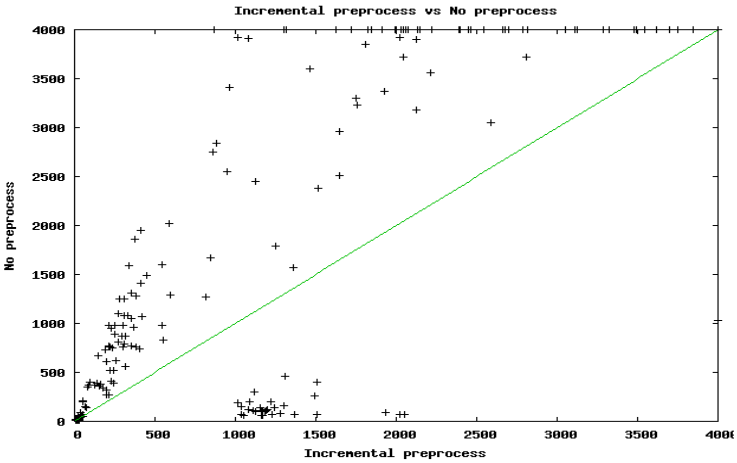


**Fig. 3.** Incremental preprocessing vs. no-preprocessing.

were solved overall faster with look-ahead, but recall that according to Fig. 1 with a long-enough timeout the two methods have very similar results in terms of the number of solved instances.

## 5 Conclusion

In various domains there is a need for incremental SAT, but the sequence of instances cannot be computed apriori, because of dependance on the result of previous instances. In such scenarios applying preprocessing with look-ahead, namely preventing elimination of variables that are expected to be reintroduced, is impossible. Incremental preprocessing, the method we suggest here, is an effective algorithm for solving this problem. Our experiments with hundreds of industrial benchmarks show that it is much faster than the two known alternatives, namely full-preprocessing and no-preprocessing. Specifically, with a timeout of 4000 sec. it is able to reduce the number of time-outs by a factor of four and three, respectively.

## References

1. F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT 2003*, volume 2919 of *LNCS*, pages 341–355, 2003.
2. Daniel Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80, 2001.
3. Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, pages 61–75, 2005.

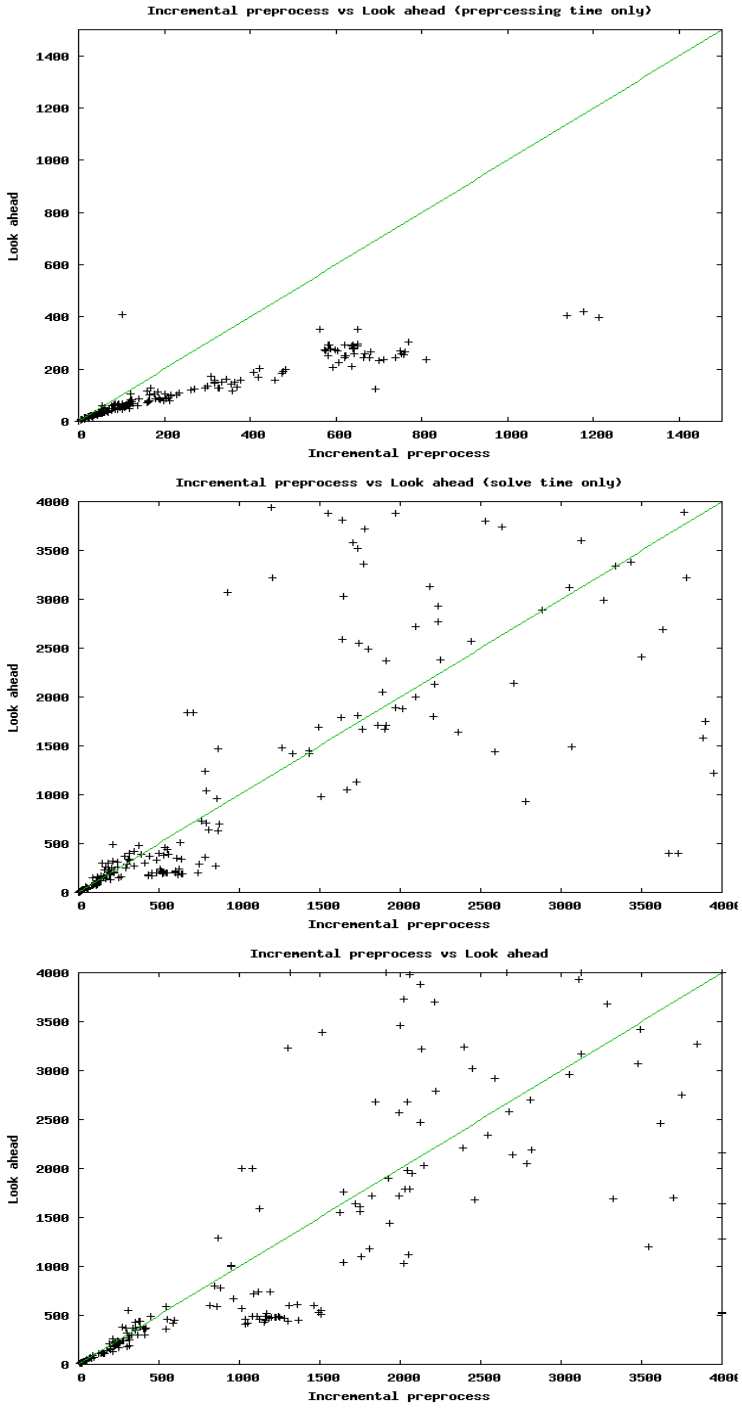


**Fig. 4.** Incremental preprocessing vs. look-ahead: (top) preprocessing time, (middle) SAT time, and (bottom) total time.

4. Niklas Eén and Niklas Sörensson. An extensible SAT-solver [ver 1.2]. In *Proceedings of Theory and Applications of Satisfiability Testing*, volume 2919 of *Lect. Notes in Comp. Sci.*, pages 512–518. Springer, 2003.
5. Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.
6. Anders Franzén, Alessandro Cimatti, Alexander Nadel, Roberto Sebastiani, and Jonathan Shalev. Applying smt in symbolic execution of microcode. In *FMCAD*, pages 121–128, 2010.
7. Stefan Kupferschmid, Matthew D. T. Lewis, Tobias Schubert, and Bernd Becker. Incremental preprocessing methods for use in bmc. *Formal Methods in System Design*, 39(2):185–204, 2011.
8. Alex Nadel, Vadim Ryvchin, and Ofer Strichman. Preprocessing in incremental sat. Technical Report IE/IS-2012-02, Industrial Engineering, Technion, 2012. Available also from `http://ie.technion.ac.il/~ofers/publications/sat12t.pdf`.
9. Ofer Shtrichman. Prunning techniques for the SAT-based bounded model checking problem. In *proc. of the 11th Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, Edinburgh, September 2001.
10. Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. SATIRE: a new incremental satisfiability engine. In *IEEE/ACM Design Automation Conference (DAC)*, 2001.