Computing prime implicants

David Déharbe[*], Pascal Fontaine[†], Daniel Le Berre[‡], Bertrand Mazure[‡]

[*] UFRN, Brazil
[†] Inria, U. of Lorraine, France
[‡] CRIL, U. of Artois, France

***Abstract*—Model checking and counter-example guided abstraction refinement are examples of applications of SAT solving requiring the production of models for satisfiable formulas. Better than giving a truth value to every variable, one can provide an implicant, i.e. a partial assignment of the variables such that every full extension is a model for the formula. An implicant is *prime* if every assignment is necessary. Since prime implicants contain no literal irrelevant for the satisfiability of the formula, they are considered as highly refined information.**

We here propose a novel algorithm that uses data structures found in modern CDCL SAT solvers to efficiently compute prime implicants starting from an existing model. The original aspects are (1) the algorithm is based on watched literals and a form of propagation of required literals, adapted to CDCL solvers (2) the algorithm works not only on clauses, but also on generalized constraints (3) for clauses and, more generally for cardinality constraints, the algorithm complexity is linear in the size of the constraints found. We implemented and evaluated the algorithm with the Sat4j library.

I. Introduction

Although SAT is a decision problem whose answer on an input formula is “satisfiable” or “unsatisfiable”, it is often necessary or useful to obtain an explanation of this output, i.e. proofs of unsatisfiability for unsatisfiable formulas and models for satisfiable formulas. As a side effect of the data structures they use, modern SAT solvers output full models for satisfiable formulas, i.e. they assign a value to every variable in the input (even if the value of some variables is irrelevant). For some applications, a partial model or implicant (i.e. a partial assignment that is sufficient to satisfy all clauses) is preferred to a full assignment. Bounded model checking is one such application: an assignment corresponds to an error trace, and the smaller the assignment, the simpler it usually is to understand the flaw [1]. Using implicants instead of models is also useful when performing Boolean optimization (e.g. Pseudo Boolean Optimization or MaxSAT). Evaluating an objective function over an implicant provides a range of values (which may contain a single element) instead of a single value with a model. As such, optimization approaches based on strengthening may compute better upper bounds from implicants rather than from models. Generating partial assignments is also useful in Satisfiability Modulo Theories (see [2] for a thorough introduction) when the theory reasoner has a high complexity. Implicants are also used in the context of compilation of knowledge base, the cover of implicants being a classical way to compile a knowledge base [3]–[5].

This work has been partially supported by CAPES grant 2347-13-0, CNPq grants 308008/2012-0, 573964/2008-4 (National Institute of Science and Technology for Software Engineering—INES), Nord-Pas de Calais Regional Council and FEDER through the ’Contrat de Projets Etat Region (CPER) 2007-2013’, and ANR TUPLES.

An implicant is *prime* if none of its proper subsets is an implicant. The paper addresses the problem of efficiently deriving a prime implicant from an existing model of a satisfiable formula. A prime implicant can be derived from a model by iteratively removing the assignments that are not necessary. In this paper, we present two instances of this greedy approach. The first associates counters to constraints, yielding the algorithm sketched in [6]. This algorithm has complexity linear in the size of the constraints, but requires specialized indexing and dedicated counters as found in DPLL-based solvers. We propose a new algorithm benefiting from the lazy data structures (i.e. watched literals [7]) available in modern SAT solvers. Our approach is not only suitable for clauses but generalizes to e.g. cardinality constraints. For sets of clauses and cardinality constraints, the complexity of this algorithm is also linear, thanks to a dedicated propagation procedure on the constraints.

Related work. We focus on computing *one* prime implicant (not necessarily of minimum size) out of a given model, using the data structures used in modern SAT solvers. Algorithm 1 (Section II-C) is quickly discussed in [6] and [8], without concrete implementation or complexity study; in Section II-C we provide a concrete instantiation of it, and discuss its complexity. An algorithm embedding SAT solving techniques is proposed in [9] and motivated by experimental results. Some other techniques, e.g. [10], involve encoding the problem of finding prime implicants to linear programming. Getting minimal assignments (i.e. prime implicants) for a CNF (Conjunctive Normal Form) from a model provided by a SAT solver is discussed in [1], and several techniques are sketched. The authors of this work notably notice that literals assigned by propagation are mandatory in any prime implicant included in the model; for completeness, we restate formally this result in Section II-C. They also mention brute-force lifting, noticing it can be implemented in time quadratic in the size of the CNF formula. In the same context, the time complexity of our algorithms is linear.

A lot of research concentrates on the problem of generating one prime implicant or the set of all prime implicants for a formula, without previous knowledge of models, e.g. [3], [8], [11]–[13]. Also, many works focus on the more complex problem of finding prime implicants of minimum size (e.g. [14] in propositional logic, and [15] in the context of SMT); the techniques presented here could be used repeatedly to find prime implicants of minimum size, but this goes beyond the scope of this paper.

Overview. Section II introduces definitions and notations. In Section II-C, we give an original formal presentation of some of the results mentioned above. Section III then presents our algorithm based on watched literals and propagation. This algorithm has been implemented in the Sat4j library [16]; experimental results are given in Section IV.

II. Basic principles

A. Definitions and notations

We assume the standard notions of propositional logic, model, propositional variable, literal and clause. A (set of) formula(s) B is a logical consequence of a (set of) formula(s) A ($A \models B$) if every model of (all elements in) A is also a model of (all elements in) B. In this paper, we use the term *constraint* for formula, implicitly understanding that a constraint c most often denotes:

- a *clause*, a disjunctive set of literals;
- a *cardinality constraint* $\sum_{\ell_i \in c} \ell_i \geq k$ where k (the degree) is an integer and each literal ℓ_i is either 0 (false) or 1 (true) — a clause can be seen as a cardinality constraint of degree 1;
- a *pseudo-Boolean constraint* $\sum_{\ell_i \in c} w_i \ell_i \geq k$, where k and each w_i are positive integers.

A set of constraints is viewed as the conjunctive combination of its elements and a *literal* as a Boolean assignment of a propositional variable. Throughout this paper, a *set of literals* cannot contain two opposite literals, so that sets of literals essentially are partial mappings from the lexicon of propositional variables to the Boolean values. In the following we identify a model for a (set of) formula(s) with the set of all the literals it satisfies.

A set of literals M is an *implicant* for a set of constraints $\mathcal{C}$ if, for every constraint $c \in \mathcal{C}$, $M \models c$. An implicant M of $\mathcal{C}$ is a *prime implicant* if, for every proper subset M' of M, M' is not an implicant of $\mathcal{C}$. Assuming $M \models c$ and $\ell \in M$, we say ℓ is a *required literal* in M for constraint c, and write $Req(M, \ell, c)$, when $M \setminus \{\ell\} \not\models c$. In particular, for a clause c such that $M \models c$, we have $Req(M, \ell, c)$ iff $M \cap c = \{\ell\}$. A *required literal* ℓ for M and a set of constraints $\mathcal{C}$, denoted $Req(M, \ell, \mathcal{C})$, is such that there exists a constraint $c \in \mathcal{C}$ with $Req(M, \ell, c)$.

B. Elements of SAT solving

Modern CDCL-based SAT solvers assume their input is given as a set of clauses, but the techniques described here may be generalized to handle cardinality and pseudo-Boolean constraints. To decide if a set of clauses is satisfiable, a solver must find a variable assignment that satisfies all clauses. Three key aspects of this search are decision, propagation and learning. *Decision* consists in setting an unassigned variable to a Boolean value. A variable assignment is *propagated* if it is enforced by the previous assignments, i.e. this happens when all but one literal in a clause have been assigned to false. Then this last literal must be true for the set of clauses to be satisfiable. It may happen that propagation implies a conflicting assignment. In that case, a new clause (the conflict) is *learnt*, being recorded as a new constraint. Then backtracking and further propagation occur. If propagation terminates without conflict, either all variables are assigned and the set is satisfiable, or a new decision occurs. On an unsatisfiable set of constraints, the algorithm will eventually reach a conflicting assignment with no decided variable.

In practice, the computation cost is dominated by propagation. A naïve algorithm could be: whenever a variable is assigned a value, all clauses containing the literal set to false are checked for unsatisfiability or new propagations. The *watched literals* technique is a heuristic that effectively reduces that cost in practice. In the case of clauses, it is based on the observation that a clause needs to be inspected only when all but one literal are assigned to false. So, for each clause, only two of its literals are watched, and the clause is inspected only when one of the two watched literals is assigned to false. This technique generalizes to cardinality constraints, by watching at most $k+1$ literals, for a constraint of degree k.

C. Greedy computation of prime implicants from models

Consider a model M for a set of constraints $\mathcal{C}$. Most often, the model M is computed with a solver using propagation; knowing which literals in M are propagated, and which are not, is highly valuable information for computing prime implicants out of M. Indeed, the following simple lemma allows to directly identify elements in M that have to be in every prime implicant included in M.

Lemma 1: Assume 1) M is an implicant for a set of formulas $\mathcal{C}$, 2) c is a logical consequence of $\mathcal{C}$, 3) and $M \setminus \{\ell\} \not\models c$. Then $M \setminus \{\ell\}$ is not an implicant of $\mathcal{C}$. In other words, the literal ℓ belongs to every prime implicant included in M.

Proof. If c is a logical consequence of $\mathcal{C}$, then every implicant of $\mathcal{C}$ is an implicant of c. As $M \setminus \{\ell\}$ is not an implicant of c, $M \setminus \{\ell\}$ is not an implicant of $\mathcal{C}$. □

In the context of CDCL solvers, the above trivial lemma has an interesting corollary. Assume $\ell \in M$ is propagated, i.e. there exists a constraint c in $\mathcal{C}$ or learnt from $\mathcal{C}$ — in both cases, c is a logical consequence of $\mathcal{C}$ — and a subset $M' \subseteq M \setminus \{\ell\}$ such that $M', c \models \ell$. Then M, $\mathcal{C}$, c and ℓ fulfill the requirements of the lemma: ℓ is mandatory in every implicant included in M. Only decision literals may possibly be removed from M to obtain a stronger implicant.

The abstract Algorithm 1 computes a prime implicant for a set of constraints $\mathcal{C}$, starting from a model M_0 of $\mathcal{C}$ and a subset Π_0 of the literals known to be in a prime implicant (e.g., the empty set, or the set of all propagated literals in the CDCL solver that produced M_0). Variable M is an implicant for $\mathcal{C}$ of decreasing size, and Π is an increasing subset of a prime implicant included in M. The algorithm checks each literal ℓ in $M \setminus \Pi$ and greedily adds it to Π if it is required or removes it from M otherwise. There may be several different prime implicants included in M_0; the successive choices of ℓ in line 3 determine which of those prime implicants is returned by the algorithm.

Algorithm 1 Abstract computation of prime implicants

```
1: procedure PRIME($\mathcal{C}, M_0, \Pi_0$)
2:     $M, \Pi \leftarrow M_0, \Pi_0$
3:     while $\ell \in M \setminus \Pi$ do
4:         if $Req(M, \ell, \mathcal{C})$ then $\Pi \leftarrow \Pi \cup \{\ell\}$
5:         else $M \leftarrow M \setminus \{\ell\}$
6:     return $\Pi$
```

The algorithm can be refined in a practical and efficient algorithm. Remember that checking if $Req(M, \ell, \mathcal{C})$ is true comes to check if $Req(M, \ell, c)$ is true for some constraint in $c \in \mathcal{C}$. It is thus useful, in order not to check every constraint in $\mathcal{C}$, to have an index $W(\ell)$ that gives the set of constraints containing ℓ. This index can be built efficiently, though it requires to read the entire set of constraints.

Algorithm 2 was sketched in [6] and is specialized for sets of clauses; it can be extended easily (at the expense of heavier notations) to cardinality constraints while preserving the linear complexity. It can also be extended to arbitrary constraints, but requires to define the concrete code for $Req(M, \ell, c)$ for an arbitrary constraint c. If c is a clause, $Req(M, \ell, c)$ is true if and only if $M \cap c = \{\ell\}$. Such a test can be done efficiently using counters for the true literals in every clause c; in Alg. 2, line 10, $\exists c \in W(\ell)$. $\mathrm{N}[c] = 1$ stands for a loop on $W(\ell)$ that stops returning true if $\mathrm{N}[c] = 1$ for some c, and returns false otherwise. For every clause c, $\mathrm{N}[c]$ has to be initialized to $|M_0 \cap c|$. The counters in N have to be updated each time a literal is removed from M (line 13).

Algorithm 2 Prime implicants for CNFs.

```
1: procedure PRIME($\mathcal{C}, M_0, \Pi_0$)
2:     $M, \Pi \leftarrow M_0, \Pi_0$
3:     for all $\ell \in M$ do $W(\ell) \leftarrow \emptyset$
4:     for all $c \in \mathcal{C}$ do
5:         $\mathrm{N}[c] \leftarrow 0$
6:         for all $\ell \in c$ do $W(\ell) \leftarrow W(\ell) \cup \{c\}$
7:     for all $\ell \in M$ do
8:         for all $c \in W(\ell)$ do $\mathrm{N}[c] \leftarrow \mathrm{N}[c] + 1$
9:     for all $\ell \in M \setminus \Pi$ do
10:        if $\exists c \in W(\ell)$ . $\mathrm{N}[c] = 1$ then
11:            $\Pi \leftarrow \Pi \cup \{\ell\}$
12:        else
13:            for all $c \in W(\ell)$ do $\mathrm{N}[c] \leftarrow \mathrm{N}[c] - 1$
14:            $M \leftarrow M \setminus \{\ell\}$
15:    return $\Pi$
```

Theorem 1: Given a satisfiable set of clauses $\mathcal{C}$, a model M_0, and a set Π_0 of literals mandatory in all prime implicants included in M_0, Algorithm 2 returns a prime implicant for $\mathcal{C}$. It runs in time $\mathcal{O}(\sum_{c\in\mathcal{C}} |c|)$.

Proof. If the set of literals given as argument of the function is a model, then the returned set of literals is also a (partial) model for $\mathcal{C}$. Indeed, a literal ℓ is removed from the model if and only if all clauses are still satisfied when ℓ is removed.

Furthermore, the returned partial model M is minimal. Assume $M \setminus \{\ell\}$ is also a partial model for $\mathcal{C}$. If ℓ has not been removed, either there exists a clause $c \in \mathcal{C}$ such that ℓ is the sole true literal, or ℓ was initially in Π_0. In the first case, $M \setminus \{\ell\}$ cannot be a partial model for c and hence for $\mathcal{C}$. The second case would contradict the theorem hypothesis on Π_0.

Assume that, for each clause c, the counter $\mathrm{N}[c]$ can be read and modified in constant time. Assume also that, for each ℓ, the indexing $W(\ell)$ of clauses containing literal ℓ is such that 1) it can be emptied in constant time, 2) an element can be added in constant time, 3) all its elements can be iteratively read in cumulative linear time. We also suppose that iterating on $\mathcal{C}$, M and $M \setminus \Pi$ has a cumulative cost which is respectively $\mathcal{O}(|\mathcal{C}|)$, $\mathcal{O}(|M|)$, and $\mathcal{O}(|M|)$.

Under the above assumptions, Algorithm 2 is linear with respect to the size of the clause set $\sum_{c\in\mathcal{C}} |c|$. We consider that every literal is present in at least one clause so that $\mathcal{O}(\sum_{c\in\mathcal{C}} |c| + |M|) = \mathcal{O}(\sum_{c\in\mathcal{C}} |c|)$. Line 3 is $\mathcal{O}(|M|)$. Lines 4–6 involve inspecting each clause and each literal in the clause, and execute a constant time operation (at line 6) for each of those literals. This block is thus $\mathcal{O}(\sum_{c\in\mathcal{C}} |c|)$. Lines 7–8 involve inspecting each clause c at most $|c|$ times, and is thus also $\mathcal{O}(\sum_{c\in\mathcal{C}} |c|)$. In the last loop at lines 9-14, each clause c from $\mathcal{C}$ is again examined at most $2 \times |c|$ times. Overall, all four loops are $\mathcal{O}(\sum_{c\in\mathcal{C}} |c|)$. □

Algorithm 2 has linear complexity, but requires to build an index of constraints by literals and counters. Also, a constraint is examined once for every of its literals satisfied in the model (not unlike what happens in SAT solving with counters instead of watched literals). Rather than preventing a counter from decreasing to 0 (which, for SAT solving, would correspond to a conflict), it is more reasonable to directly put in Π the last satisfied literal of a clause as soon as the counter reaches one (i.e. using some kind of propagation). This motivates the version presented in the next section, using watched literals, instead of indexes and counters.

III. Computing prime implicants by propagation

It can be argued that the above algorithm uses late detection of literals to add in the prime implicant. Indeed, a literal ℓ is iteratively selected, and $Req(M, \ell, c)$ is checked for every constraint c containing ℓ. Another possibility is to use early detection of literals for addition to Π, similarly to Boolean constraint propagation in SAT solvers. This yields the algorithms described in this section.

A. An abstract version

Algorithm 3 is the early detection equivalent of abstract Algorithm 1: it computes a prime implicant out of an implicant

M_0 for a set of constraints $\mathcal{C}$, and any subset Π_0 of the required literals in M_0. Variable M, initialized to M_0, is an implicant for $\mathcal{C}$ of strictly decreasing size, and Π is an increasing subset of a prime implicant included in M. The larger Π_0 is, the faster the convergence[1]; also it is optional as the algorithm is sound if it is empty. We introduce it for future specializations.

Algorithm 3 Abstract propagation-based algorithm

```
1: procedure PRIME($\mathcal{C}, M_0, \Pi_0$)
2:     $M, \Pi \leftarrow M_0, \Pi_0$
3:     $\Pi \leftarrow \Pi \cup$ IMPLIED($\mathcal{C}, M$)
4:     while $\ell \in M \setminus \Pi$ do
5:         $M \leftarrow M \setminus \{\ell\}$
6:         $\Pi \leftarrow \Pi \cup$ IMPLIED($\mathcal{C}, M$)
7:     return $\Pi$
```

Algorithm 3 first identifies and adds to Π the required literals of M (l. 3). Repeatedly one of the remaining literals in $M \setminus \Pi$ is removed (l. 4) until $M \setminus \Pi$ is empty. This may trigger other remaining literals to be added to Π (l. 6). The call IMPLIED$(\mathcal{C}, M)$ yields a subset of M such that

$$\text{IMPLIED}(\mathcal{C}, M) \setminus \Pi = \big\{\ell \mid \mathit{Req}(M, \ell, \mathcal{C})\big\} \setminus \Pi,$$

i.e. IMPLIED$(\mathcal{C}, M)$ returns the set of literals in M that should be added to Π because, for each of these literals, a constraint requires this literal to be true. Note that, in contrast to Algorithm 1, the literal chosen in l. 4 is removed from the prime implicant without further test. Lines 3 and 6 establish the property that no literal in $M \setminus \Pi$ is required.

Proposition 1: Given a set of constraints $\mathcal{C}$, an implicant M_0, and Π_0, a subset of $\{\ell \mid \mathit{Req}(M_0, \ell, \mathcal{C})\}$, Algorithm 3 terminates and returns a prime implicant of $\mathcal{C}$ included in M_0.
Proof. The loop in Algorithm 3 satisfies the following invariants:

I_1: $\Pi = \{\ell \mid \mathit{Req}(M, \ell, \mathcal{C})\}$;
I_2: $\Pi \subseteq M \subseteq M_0$;
I_3: M is an implicant.

Invariant I_1 is verified at the start of the loop as a consequence of line 3 (assuming the pre-condition $\Pi_0 \subseteq \{\ell \mid \mathit{Req}(M, \ell, \mathcal{C})\}$ for the call to PRIME) and is preserved thanks to line 6. I_2 is trivial, and I_3 is verified at the start of the loop as a consequence of line 2. It is preserved since ℓ at lines 4 and 5 is not in Π, thus is not required: $\forall c \,.\, M \setminus \{\ell\} \models c$. The new value of M is again an implicant for all c.

The loop variant $|M \setminus \Pi|$ is a strictly decreasing sequence of natural numbers; the loop terminates when $M \subseteq \Pi$, i.e. when $M = \Pi$ (thanks to invariant I_2) and $\Pi \subseteq M_0$. From invariant I_3, Π is an implicant and from invariant I_1, this implicant is prime. This establishes the property.

The above proof is suitable for any type of Boolean constraints. For the special case of a clause $c \in \mathcal{C}$, notice that, as a direct consequence of the loop invariant, $c \cap \Pi \neq \emptyset \vee |c \cap M| \geq 2$. $\square$

[1]Technically, a SAT solver should assign Π_0 to the set of literals assigned by unit propagation while establishing $M_0 \models \mathcal{C}$.

There may exist several prime implicants in M_0. The one produced by Algorithm 3 depends only on the successive choices of ℓ in line 4. Any prime implicant subset of M_0 may be produced, given the right sequence of chosen literals. A prime implicant produced by Algorithm 1 or Algorithm 2 is obtained by Algorithm 3 by picking literals in the same sequence and dropping literals that are already in prime.

B. Implementation with watched literals

A concrete implementation of the above abstract algorithm would best use the data structures implemented in state-of-the-art SAT solvers. This is the approach of Algorithm 4: in addition to the model M_0, it reuses the watched literals relation at the ending state of the SAT solver. We consider a general notion of watched literals as a relation W between literals and constraints such that, for every literal ℓ, $W(\ell)$ is a (sub)set of constraints containing ℓ. We now require Π_0 to initially contain all the literals that are directly entailed by one constraint in $\mathcal{C}$.[2] Since such literals are included in $\{\ell \mid \mathit{Req}(M, \ell, \mathcal{C})\}$ the precondition for Algorithm 3 is verified.

Algorithm 4 Prime implicants using watched literals

```
1: procedure PRIME($\mathcal{C}, M_0, \Pi_0, W$)
2:     $M, \Pi \leftarrow M_0, \Pi_0$
3:     IMPLIED$_{W,0}$($\mathcal{C}, M, \Pi, W$)
4:     while $\ell \in M \setminus \Pi$ do
5:         $M \leftarrow M \setminus \{\ell\}$
6:         IMPLIED$_W$($\mathcal{C}, M, \ell, \Pi, W$)
7:     return $\Pi$

8: procedure IMPLIED$_{W,0}$($\mathcal{C}, M$, ref $\Pi$, ref $W$)
9:     for all $\ell \in M \setminus \Pi$ do
10:        IMPLIED$_W$($\mathcal{C}, M, \bar{\ell}, \Pi, W$)

11: procedure IMPLIED$_W$($\mathcal{C}, M, \ell$, ref $\Pi$, ref $W$)
12:     $W_\ell \leftarrow W(\ell)$
13:     for all $c \in W_\ell$ do
14:         HDL_CONSTR($c, M, \ell, \Pi, W$)
```

The data in Algorithm 4 includes the variables of Algorithm 3, namely M and Π, and the watched literals relation W. The inherent property of the watched literals for a constraint c, i.e. $W^{-1}(c)$, is that, as long as all watched literals remain either undefined or true, nothing can be deduced from c in the current assignment. In our context Π plays a role similar to the current partial assignment in the SAT solver. Let us define, for a constraint c, the set of literals $S(c) = \Pi \cup W^{-1}(c)$. Formally, W is always such that:

$W_1(c)$: $\forall \ell \in W^{-1}(c) \setminus \Pi \,.\, \neg \mathit{Req}(S(c), \ell, c)$
$W_2(c)$: $S(c) \cap M \models c$

Both properties should be satisfied by the inputs given to our algorithm. Observe that: 1) if $\Pi \models c$, then $W_1(c)$ is true; 2)

[2]In particular, Π_0 should contain all literals in unit clauses.

when a literal ℓ is removed from M, $W_1(c)$ is not affected; and 3) if ℓ furthermore satisfies $\neg Req(M, \ell, c)$, then $W_2(c)$ is also preserved. The algorithm first establishes an additional loop invariant (line 3):

$W_3(c)$: $S(c) \subseteq M$

As in Algorithm 3, the main loop repeatedly removes one (unrequired) literal ℓ from $M \setminus \Pi$ (line 5), possibly augmenting Π with new literals, and repairs the invariant properties for the watched literals (line 6). Function HDL_CONSTR (Algorithm 5) reestablishes these properties for each c affected by the removal of ℓ from M. Its definition is left general enough so that it can be specialized for different classes of constraints and watched literals strategies. This greedy approach is similar to Boolean propagation in SAT solving, Π emulating the assignment whereas M restricts the choice for watched literals and possible propagations.

Algorithm 5 HDL_CONSTR for arbitrary constraints

```
1: procedure HDL_CONSTR($c, M, \ell$, ref $\Pi$, ref $W$)
2:     $\Pi \leftarrow \Pi \cup \{\ell' \in W^{-1}(c) \mid Req(M, \ell', c)\}$
3:     if $\Pi \not\models c$ then
4:         Choose $W'$ such that
5:             $W' \subseteq (W^{-1}(c) \cup M) \setminus \{\ell\}$
6:             $(\Pi \cup W') \cap M \models c$
7:             $\forall \ell' \in W' \setminus \Pi \,.\, \neg Req(W' \cup \Pi, \ell', c)$
8:         in $W^{-1}(c) \leftarrow W'$
```

Proposition 2: Given a set of constraints $\mathcal{C}$, an implicant M_0, a set of literals Π_0, and a relation W between literals and constraints in $\mathcal{C}$ such that:

- $\{\ell \mid \exists c \in \mathcal{C} \,.\, c \models \ell\} \subseteq \Pi_0 \subseteq \{\ell \mid Req(M_0, \ell, \mathcal{C})\}$,
- $\forall c \in \mathcal{C} \,.\, W_1(c) \wedge W_2(c)$

then Algorithm 4 terminates and returns a prime implicant of $\mathcal{C}$ contained in M_0.

Proof. The proof is similar to that of Algorithm 3, the same invariants being satisfied: we prove that lines 3 and 6 establish these invariants through the successive calls to function HDL_CONSTR (Algorithm 5).

First, consider the call in line 3. Before the call, invariants I_2 and I_3 are satisfied, as well as $W_1(c)$ and $W_2(c)$ for each constraint c, and $\Pi \subseteq \{\ell \mid Req(M_0, \ell, \mathcal{C})\}$, thanks to the preconditions of PRIME. The call establishes $W_3(c)$ for every constraint c, and at the same time, introduces literals in Π so that $\{\ell \mid Req(M_0, \ell, \mathcal{C})\} \subseteq \Pi$. This is a direct consequence of line 2 in Algorithm 5, the other lines ensuring that $W_1(c)$, and $W_2(c)$ remain preserved even if $\bar{\ell}$ is removed. When all the negations of literals in M have been removed from the watched literals by the successive calls, $W_3(c)$ is also satisfied for each c. Every element ℓ added in Π can be related to a constraint c such that $Req(M, \ell, c)$.

Now consider the call in l. 6. The invariants are satisfied, if it were not for the absence of ℓ in M. Again, for each constraint c, the successive calls to HDL_CONSTR repair the invariant $W_3(c)$ while preserving $W_1(c)$ and $W_2(c)$. This may insert new literals in Π if they are required by c. □

Algorithm 6 HDL_CONSTR for clause or cardinality constraints

```
1: procedure HDL_CONSTR($c, M, \ell$, ref $\Pi$, ref $W$)
2:     if $\exists \ell' \in c \cap M \,.\, \ell' \notin W^{-1}(c)$ then
3:         $W \leftarrow (W \cup \{\ell' \mapsto c\}) \setminus \{\ell \mapsto c\}$
4:     else $\Pi \leftarrow \Pi \cup (W^{-1}(c) \setminus \{\ell\})$
```

Function HDL_CONSTR in Algorithm 5 is generic and may be refined to handle specific classes of constraints. One such concrete implementation is given for clauses and, more generally, for cardinality constraints in Algorithm 6. Assuming $|W^{-1}(c)| \geq 2$, $c \in W(\ell)$ and $\ell \notin M$, either there exists another literal ℓ' that may be watched by c, in which case W is updated with the new association, or there is no such literal, and the literals in $W^{-1}(c)$ must be in the prime implicant and are inserted into Π. In the special case of clauses, then there is only one such literal.

Proposition 3: When $\mathcal{C}$ is a set of clauses and HDL_CONSTR is specified as in Algorithm 6, Algorithm 4 runs in time $\mathcal{O}(\sum_{c\in\mathcal{C}} |c|)$.

Proof. IMPLIED$_W(\mathcal{C}, M, \ell, \Pi, W)$ has cumulated complexity in $\mathcal{O}(\sum_{c\in\mathcal{C}} |c|)$. To achieve this rate, one has to ensure that, for every clause c, the cumulative time for the calls to HDL_CONSTR with c is $\mathcal{O}(|c|)$. This can simply be done by storing clauses as arrays of literals indexed from 1 to $|c|$, using a pointer initialized to 1 in this array, and looking for the suitable literal from this pointer on (and updating its value). The successive calls to HDL_CONSTR on clause c would then resume their search from the previous position. Each literal in each clause would therefore be processed at most once.

For every literal ℓ in M, there is one call to IMPLIED$_W(\mathcal{C}, M, \bar{\ell}, \Pi, W)$ in function IMPLIED$_{W,0}$. Every clause in $\mathcal{C}$ is satisfied by at least one of its watched literals. If a clause appears in $W(\bar{\ell})$, its other watched literal is thus in M, and the watched $\bar{\ell}$ will be replaced by another watched literal in M (or the clause stays in $W(\bar{\ell})$ and its other watched literal is added to Π). So every clause will be examined at most once for the whole run of IMPLIED$_{W,0}$. Assuming the search for another watched literal in line 2 of Algorithm 6 remains linear with respect to the size of the clause, IMPLIED$_{W,0}$ runs in time $\mathcal{O}(\sum_{c\in\mathcal{C}} |c|)$.

IMPLIED$_W$ is called at most once for each literal ℓ in M_0 on line 6 in Algorithm 4. If the watch relation for clause c is modified (on line 3 in Algorithm 6), c will never occur again in $W(\ell)$, since ℓ is removed forever from M. As a consequence, every clause c will be considered at most $|c|$ times by the successive calls of Algorithm 6. In these calls, the cumulated searches for a new watched literal (condition on line 2 in Alg. 6) accounts for a factor linear in the size of c. □

Prop. 3 may be generalized to any class of constraints and watched literals strategy where the cumulated time of HDL_CONSTR(c) has a complexity linear in the size of c. This holds for cardinality constraints, as the watched literals strategy may also be employed.

IV. Experimental evaluation

A classical implementation of Algorithm 1 with quadratic complexity in the size of $M_0 \setminus \Pi_0$ has already been available in Sat4j for several years. In practice, this implementation performed well on many SAT benchmarks because a vast majority of the literals of the model found by the SAT solver are implied by unit propagation, so $M_0 \setminus \Pi_0$ was initially much smaller than M_0 (Π_0 containing all propagated literals initially). There are however classes of problems for which this is not true.

Sat4j MaxSAT uses selector variables to translate MaxSAT problems into Pseudo-Boolean Optimization problems [16]. In that context, counting the number of satisfied selector variables provides an upper bound on the minimum number of constraints that must be falsified. However, despite a strategy to always branch first on falsified selector variables, some selector variables may be satisfied even if the original constraint is satisfied. To improve the bounds, two solutions exist: using an encoding enforcing that the selector variable can only be satisfied if the original constraint is falsified, or counting the selector variables on a prime implicant. The former solution adds many binary clauses to the original CNF (as many binary clauses as literals in the original formula) and is inefficient in practice.

We present here some experimental results of the proposed algorithm on a specific set of benchmarks from the MaxSAT 2010 evaluation. The previous version of Sat4j could not compute prime implicants for industrial MaxSAT benchmarks from *circuit debugging* with millions of variables and clauses [17]. We used those benchmarks to compare the proposed algorithm based on watched literals against one based on counters, both of linear complexity.

Since we use prime implicants to improve the upper bounds computed by our MaxSAT solver, prime implicants have to be computed on a set of clauses plus one cardinality or pseudo-boolean constraint representing the bound of the objective function. On the following, we present the time required to compute the first prime implicant of each benchmark, thus on clauses only.

Algorithm 4 (for clauses) and Algorithm 6 (for clauses, cardinality and pseudo-Boolean constraints) have been implemented in the Sat4j library. As described in the previous section, the implementation includes a propagation procedure similar to the classical unit propagation scheme found in CDCL solvers with two key differences: i) the propagation always eventually finds a satisfied literal and ii) the number of steps to update the watched literals is reduced by storing the last position in the search between each call to the propagation procedure. Note that for clauses, where only two literals are watched, a constraint with n literals is traversed at most n times if there is no bookkeeping, and it may be a good tradeoff to avoid storing that information for short clauses to save memory. For larger clauses, or cardinality constraints, bookkeeping the state of the search as proposed for Algorithm 6 is crucial: on some examples, the time spent to compute a prime implicant was dramatically reduced (e.g. from 240 seconds to less than one second) by such a simple implementation detail, that guarantees the linearity of the algorithm. For pseudo-Boolean constraints, we use a counter based implementation and extra care is required to update the state during backtracking and to handle the literals that do not belong to the implicant. Finally, learned clauses are ignored for the propagation. The implementation details can be found in the source code of Sat4j.

The results are summarized in Table I. The Sean Safarpour benchmark set contains 52 benchmarks. Sat4j is able to load 36, running out of memory for the others (when given 2GB of memory). For those 36 benchmarks, we give the number of variables (including the selector variables, one per clause), the number of clauses, the total number of literals in the formula (the cumulated size of the clauses), the number of literals implied by unit propagation in the model (#implied), and the time taken respectively by the counter *vs.* watched literals approaches to compute a prime implicant from the first model found by the MaxSAT solver. We also give the median values on those 36 benchmarks. The proposed algorithm was able to compute prime implicants for all benchmarks within a second, while the counter based approach missed one (due to memory out) and lead in some cases to much greater runtimes (up to one order of magnitude). Those results illustrate the advantage of reusing the solver data structures to minimize memory requirements and the advantage in practice of using those lazy data structure for computing prime implicants.

V. Conclusion

We propose and discuss an algorithm to compute prime implicants in time linear in the size of the input formula designed for easy integration in modern SAT solvers. This algorithm is based on lazy data structures such as watched literals [7]. The efficiency of the algorithm is maintained for other kinds of constraints as long as some data structure ensures the constraint will be traversed at most once during the successive calls to the propagation procedure. This applies to both clauses and cardinality constraints. The same algorithm can also be applied to other kind of constraints, but linear complexity may be lost.

We implemented the algorithm for clauses, cardinality and pseudo-Boolean constraints in the Sat4j platform. On a class of problems with millions of variables, we compare a counter based algorithm against our watched literal algorithm. While both algorithms are linear, our algorithm computed all prime implicants in less than a second, which was not the case for the other algorithm. These results show that applying the proposed algorithm to compute prime implicants instead of models has a negligible overhead.

Good prime implicant computation procedures are useful for many applications. In particular, we investigate prime implicants for Boolean optimization by strengthening, as the value of the objective function computed on a prime implicant

TABLE I
EXPERIMENTAL COMPARISON OF THE PRIME IMPLICANT ALGORITHMS ON SELECTED SEAN SAFARPOUR BENCHMARKS (2GB MEMORY).

#vars (M)	#cla (M)	#literals (M)	#implied (M)	Alg. 2 (s)	Alg. 4 (s)
2.3	1.7	4.0	0.5	4.842	**0.736**
1.3	0.9	2.2	0.4	**0.347**	0.377
1.5	1.1	2.7	0.4	2.860	**0.495**
2.6	1.8	4.4	0.6	MO	**3.463**
1.5	1.0	2.5	0.3	0.541	**0.380**
0.7	0.5	1.3	0.2	**0.210**	0.230
0.7	0.5	1.3	0.2	**0.212**	0.237
1.0	0.7	1.8	0.3	0.729	**0.364**
0.9	0.7	1.8	0.2	**0.225**	0.252
1.0	0.7	1.9	0.2	0.559	**0.283**
1.0	0.7	1.9	0.2	0.552	**0.283**
1.0	0.8	2.1	0.2	0.578	**0.301**
0.2	0.16	0.4	0.04	0.154	**0.120**
0.5	0.4	1.1	0.1	0.552	**0.221**
0.2	0.9	2.4	0.25	**0.280**	0.353
2.0	1.5	3.9	0.5	4.191	**0.486**
1.6	1.2	2.9	0.4	3.956	**0.377**
1.0	0.8	2.1	0.2	0.638	**0.284**
1.8	1.0	2.8	0.3	4.008	**0.354**
2.0	1.6	4.5	0.4	2.567	**0.486**
1.1	0.9	2.6	0.2	0.326	**0.304**
1.1	0.9	2.6	0.2	0.333	**0.289**
1.1	0.9	2.6	0.2	**0.319**	0.330
1.1	0.9	2.6	0.2	**0.343**	0.684
2.0	1.6	4.6	0.4	2.493	**0.493**
0.8	0.7	1.9	0.1	**0.232**	0.269
1.2	0.9	2.5	0.2	0.621	**0.348**
0.2	0.1	0.3	0.04	0.152	**0.102**
0.2	0.1	0.3	0.04	0.154	**0.077**
2.2	1.7	4.8	0.4	9.225	**0.510**
2.2	1.7	4.8	0.4	8.946	**0.490**
2.2	1.7	4.8	0.4	6.086	**0.556**
1.5	1.2	3.4	0.3	4.250	**0.366**
1.5	1.2	3.4	0.3	4.172	**0.370**
1.0	0.8	1.9	0.3	0.643	**0.285**
1.0	0.8	1.9	0.3	0.645	**0.273**
Median					
1.168	0.930	-	0.268	0.578	0.301

yields a better upper bound than the value obtained with a model.

REFERENCES

[1] K. Ravi and F. Somenzi, "Minimal assignments for bounded model checking," in *TACAS*, 2004, vol. 2988 of *LNCS*.

[2] C. Barrett, R. Sebastiani, S.A. Seshia, and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Satisfiability*, vol. 185 of *Frontiers in Artificial Intelligence and Applications*. 2009.

[3] R. Schrag, "Compilation for critically constrained knowledge bases," in *AAAI*, 1996, pp. 510–515.

[4] Y. Boufkhad, É. Grégoire, P. Marquis, B. Mazure, and L. Saïs, "Tractable cover compilations," in *IJCAI*, 1997.

[5] A. Darwiche and P. Marquis, "A knowledge compilation map," *J. Artificial Intelligence Research*, vol. 17, 2002.

[6] T. Castell, "Computation of prime implicates and prime implicants by a variant of the Davis and Putnam procedure," in *ICTAI*, 1996.

[7] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *DAC*, 2001.

[8] L. Palopoli, F. Pirri, and C. Pizzuti, "Algorithms for selective enumeration of prime implicants," *Artif. Intell.*, vol. 111, no. 1-2, 1999.

[9] S. Shen, Y. Qin, and S. Li, "Minimizing counterexample with unit core extraction and incremental SAT," in *VMCAI*, R. Cousot, Ed., 2005, vol. 3385 of *LNCS*.

[10] C. Pizzuti, "Computing prime implicants by integer programming," in *ICTAI*, 1996.

[11] A. Kean and G. Tsiknis, "An incremental method for generating prime implicants/implicates," *J. Symbolic Computation*, vol. 9, no. 2, 1990.

[12] N. V. Murray and E. Rosenthal, "Linear response time for implicate and implicant queries," *Knowl. Inf. Syst.*, vol. 22, no. 3, 2010.

[13] A. Ramesh, G. Becker, and N. V. Murray, "CNF and DNF considered harmful for computing prime implicants/implicates," *J. Automated Reasoning*, vol. 18, no. 3, 1997.

[14] V. Manquinho, P. Flores, J. P. Marques Silva, and A. Oliveira, "Prime implicant computation using satisfiability algorithms," in *ICTAI*, 1997.

[15] I. Dillig, T. Dillig, K. L. McMillan, and A. Aiken, "Minimum satisfying assignments for SMT," in *Computer Aided Verification (CAV)*, 2012, vol. 7358 of *LNCS*.

[16] D. Le Berre and A. Parrain, "The Sat4j library, release 2.2," *JSAT*, vol. 7, no. 2-3, 2010.

[17] S. Safarpour, H. Mangassarian, A. Veneris, M. Liffiton, and K. Sakallah, "Improved design debugging using maximum satisfiability," in *FMCAD*, 2007.