Efficient CNF Simplification based on Binary Implication Graphs*

Marijn Heule[1], Matti Järvisalo[2], and Armin Biere[3]

[1] Department of Software Technology, Delft University of Technology, The Netherlands
[2] Department of Computer Science, University of Helsinki, Finland
[3] Institute for Formal Models and Verification, Johannes Kepler University Linz, Austria

Abstract. This paper develops techniques for efficiently detecting redundancies in CNF formulas. We introduce the concept of *hidden literals*, resulting in the novel technique of *hidden literal elimination*. We develop a practical simplification algorithm that enables "*Unhiding*" various redundancies in a unified framework. Based on time stamping literals in the binary implication graph, the algorithm applies various binary clause based simplifications, including techniques that, when run repeatedly until fixpoint, can be too costly. *Unhiding* can also be applied during search, taking learnt clauses into account. We show that *Unhiding* gives performance improvements on real-world SAT competition benchmarks.

1 Introduction

Applying reasoning techniques (see e.g. [1,2,3,4,5,6,7]) to simplify Boolean satisfiability (SAT) instances both before and during search is important for improving state-of-the-art SAT solvers. This paper develops techniques for efficiently detecting and removing redundancies from CNF (conjunctive normal form) formulas based on the underlying *binary clause structure* (i.e., the binary implication graph) of the formulas.

In addition to considering known simplification techniques (hidden tautology elimination (HTE) [6], hyper binary resolution (HBR) [1,7], failed literal elimination over binary clauses [8], equivalent literal substitution [8,9,10], and transitive reduction [11] of the binary implication graph [10]), we introduce the novel technique of *hidden literal elimination* (HLE) that removes so-called *hidden literals* from clauses without affecting the set of satisfying assignments. We establish basic properties of HLE, including conditions for achieving confluence when combined with equivalent literal substitution.

As the second main contribution, we develop an efficient and practical simplification algorithm that enables "*Unhiding*" various redundancies in a unified framework. Based on time stamping literals via randomized depth-first search (DFS) over the binary implication graph, the algorithm provides efficient approximations of various binary clause based simplifications which, when run repeatedly until fixpoint, can be too costly. In particular, while our *Unhiding* algorithm is linear time in the total number of literals (with an at most logarithmic factor in the length of the longest clause), notice as an example that fixpoint computation of failed literals, even just on the binary implication

* The 1st author is financially supported by Dutch Organization for Scientific Research (grant 617.023.611), the 2nd author by Academy of Finland (grant 132812) and the 1st and 3rd author are supported by the Austrian Science Foundation (FWF) NFN Grant S11408-N23 (RiSE).

graph, is conjectured to be at least quadratic in the worst case [8]. *Unhiding* can be implemented without occurrence lists, and can hence be applied not only as a preprocessor but also *during search*, which allows to take learnt clauses into account. Indeed, we show that, when integrated into the state-of-the-art SAT solver Lingeling [12], *Unhiding* gives performance improvements on real-world SAT competition benchmarks.

On related work, Van Gelder [8] studied exact and approximate DFS-based algorithms for computing equivalent literals, failed literals over binary clauses, and implied (transitive) binary clauses. The main differences to this work are: (i) *Unhiding* approximates the additional techniques of HTE, HLE, and HBR; (ii) the advanced DFS-based time stamping scheme of *Unhiding* detects failed and equivalent literals *on-the-fly*, in addition to *removing* (instead of adding as in [8]) transitive edges in the binary implication graph; and (iii) *Unhiding* is integrated into a clause learning (CDCL) solver, improving its performance on real application instances (in [8] only random 2-SAT instances were considered). Our advanced stamping scheme can be seen as an extension of the BinSATSCC-1 algorithm in [13] which excludes (in addition to cases (i) and (iii)) transitive reduction. Furthermore, while [13] focuses on simplifing the binary implication graph, we use reachability information obtained from traversing it to simplify larger clauses, including learnt clauses, in addition to extracting failed literals.

As for more recent developments, CryptoMiniSAT v2.9.0 [14] caches implied literals, and updates the cache after top-level decisions. The cache can serve a similar purpose as our algorithms, removing literals and clauses. Yet, the cache size is quadratic in the number of literals, which is also the case for using the cache for redundancy removal for the whole CNF. Thus, at least from a complexity point of view, the cache of CryptoMiniSAT does not improve on the quadratic algorithm [8]. In contrast, *Unhiding* requires only a single sweep over the binary implication graph and the other clauses.

After preliminaries (CNF satisfiability and known CNF simplification techniques, Sect. 2), we introduce hidden literal elimination and establish its basic properties (Sect. 3). We then explain the *Unhiding* algorithm: basic idea (Sect. 4) and integration of simplification techniques (Sect. 5). Then we develop an advanced version of *Unhiding* that can detect further redundancies (Sect. 6), and present experimental results (Sect. 7).

2 Preliminaries

For a Boolean variable x, there are two *literals*, the positive literal x and the negative literal $\bar{x}$. A *clause* is a disjunction of literals and a CNF formula a conjunction of clauses. A clause can be seen as a finite set of literals and a CNF formula as a finite set of clauses. A truth assignment for a CNF formula F is a function τ that maps literals in F to $\{0,1\}$. If $\tau(x) = v$, then $\tau(\bar{x}) = 1 - v$. A clause C is satisfied by τ if $\tau(l) = 1$ for some literal $l \in C$. An assignment τ satisfies F if it satisfies every clause in F.

Two formulas are *logically equivalent* if they are satisfied by exactly the same set of assignments. A clause is a *tautology* if it contains both x and $\bar{x}$ for some variable x. The length of a clause is the number of literals in the clause. A clause of length one is a *unit clause*, and a clause of length two is a *binary clause*. For a CNF formula F, we denote the set of binary clauses in F by F_2.

Binary Implication Graphs For any CNF formula F, we associate a unique directed *binary implication graph* $\mathsf{BIG}(F)$ with the edge relation $\{\langle\bar{l}, l'\rangle, \langle\bar{l}', l\rangle \mid (l \vee l') \in F_2\}$.

In other words, for each binary clause $(l \vee l')$ in F, the two implications $\bar{l} \rightarrow l'$ and $\bar{l}' \rightarrow l$, represented by the binary clause, occur as edges in $\mathsf{BIG}(F)$. A node in $\mathsf{BIG}(F)$ with no incoming arcs is a *root* of $\mathsf{BIG}(F)$ (or, simply, of F_2). In other words, literal l is a root in $\mathsf{BIG}(F)$ if there is no clause of the form $(l \vee l')$ in F_2. The set of roots of $\mathsf{BIG}(F)$ is denoted by $\mathrm{RTS}(F)$.

2.1 Known Simplification Techniques

BCP and Failed Literal Elimination (FLE**)** For a CNF formula F, *Boolean constraint propagation* (BCP) (or *unit propagation*) propagates all unit clauses, i.e. repeats the following until fixpoint: if there is a unit clause $(l) \in F$, remove from $F \setminus \{(l)\}$ all clauses that contain the literal l, and remove the literal $\bar{l}$ from all clauses in F, resulting in the formula $\mathrm{BCP}(F)$. A literal l is a *failed literal* if $\mathrm{BCP}(F \cup \{(l)\})$ contains the empty clause, implying that F is logically equivalent to $\mathrm{BCP}(F \cup \{(\bar{l})\})$. FLE removes failed literals from a formula, or, equivalently, adds the complements of failed literals as unit clauses to the formula.

Equivalent Literal Substitution (ELS**)** The strongly connected components (SCCs) of $\mathsf{BIG}(F)$ describe equivalent classes of literals (or simply equivalent literals) in F_2. *Equivalent literal substitution* refers to substituting in F, for each SCC G of $\mathsf{BIG}(F)$, all occurrences of the literals occurring in G with the representative literal of G. ELS is confluent, i.e., has a unique fixpoint, modulo variable renaming.

Hidden Tautology Elimination (HTE**)** [6] For a given CNF formula F and clause C, (*hidden literal addition*) $\mathrm{HLA}(F, C)$ is the *unique* clause resulting from repeating the following clause extension steps until fixpoint: if there is a literal $l_0 \in C$ such that there is a clause $(l_0 \vee l) \in F_2 \setminus \{C\}$ for some literal l, let $C := C \cup \{\bar{l}\}$. Note that $\mathrm{HLA}(F, C) = \mathrm{HLA}(F_2, C)$. Further, for any $l \in \mathrm{HLA}(F, C) \setminus C$, there is a path in $\mathsf{BIG}(F)$ from l to some $l_0 \in C$. For any CNF formula F and clause $C \in F$, $(F \setminus \{C\}) \cup \{\mathrm{HLA}(F, C)\}$ is logically equivalent to F [6]. Intuitively, each extension step in computing HLA is an application of self-subsuming resolution [2,15,16] in reverse order. For a given CNF formula F, a clause $C \in F$ is a *hidden tautology* if and only if $\mathrm{HLA}(F, C)$ is a tautology. *Hidden tautology elimination* removes hidden tautologies from CNF formulas.

Note that *distillation* [4] is more generic than HTE [6] (and also more generic than HLE as defined in this paper). However, it is rather costly to apply, and is in practice restricted to irredundant/original clauses only.

Transitive reduction of the binary implication graph (TRD**)** A directed acyclic graph G' is a *transitive reduction* [11] of the directed graph G provided that (i) G' has a directed path from node u to node v if and only if G has a directed path from node u to node v, and (ii) there is no graph with fewer edges than G' satisfying condition (i). It is interesting to notice that, by applying FLE restricted to the literals in F_2 before HTE, HTE achieves a transitive reduction of $\mathsf{BIG}(F)$ for any CNF formula F purely on the clausal level [6].

3 Hidden Literal Elimination

In this section we present a novel redundancy elimination procedure exploiting the binary clause structure of a CNF formula. We call the technique *hidden literal elimination*.

For a given CNF formula F and literal l, we denote by $\mathrm{HL}(F, l)$ the *unique* set of *hidden literals* of l w.r.t F. $\mathrm{HL}(F, l)$ is defined as follows. First, let $L = \{l\}$. Then repeat the following steps until fixpoint: if there is a literal $l_0 \in L$ such that there is a clause $(l_0 \vee l') \in F_2$ for some literal l', let $L := L \cup \{\bar{l'}\}$. Now, let $\mathrm{HL}(F, l) := L \setminus \{l\}$. In other words, $\mathrm{HL}(F, l)$ contains the complements of all literals that are reachable from $\bar{l}$ in $\mathsf{BIG}(F)$, or, equivalently, all literals from which l is reachable in $\mathsf{BIG}(F)$. Notice that $\mathrm{HL}(F, l) = \mathrm{HL}(F_2, l)$. Also, HL captures failed literals in F_2 in the sense that by definition, for any literal l in F_2, there is a path from l to $\bar{l}$ in $\mathsf{BIG}(F)$ if and only if $\bar{l} \in \mathrm{HL}(F, l)$.

Proposition 1. *For any CNF formula F, a literal l in F_2 is failed iff $\bar{l} \in \mathrm{HL}(F, l)$.*

For a given formula F, *hidden literal elimination* (HLE) repeats the following: if there is a clause $C \in F$ and a literal $l \in C$ such that $C \cap \mathrm{HL}(F, l) \neq \emptyset$, let $F := (F \setminus \{C\}) \cup \{C \setminus \mathrm{HL}(F, l)\}$. In fact, the literals in $\mathrm{HL}(F, l)$ can be removed from all clauses that contain l.

Proposition 2. *For every CNF formula F, any result of applying* HLE *on F is logically equivalent to F.*

Proof. For any CNF formula F and two literals l and l', if $l' \in \mathrm{HL}(F, l)$, then $F \cup \{(l')\}$ logically implies l by the definition of HL. Hence, for any clause $C \in F$ with $l, l' \in C$, for any satisfying assignment τ for F with $\tau(l') = 1$ we have $\tau(l) = 1$, and hence τ satisfies $(F \setminus \{C\}) \cup \{C \setminus \mathrm{HL}(F, l)\}$. □

A relevant question is how many literals HLE eliminates relative to other literal elimination techniques. One example is self-subsuming resolution (SSR) [2] that replaces clauses that have a resolvent that subsumes the clause itself with the resolvent (essentially eliminating from the clause the literal not in the resolvent).

Proposition 3. *There are CNF formulas from which* HLE *can remove more literals from clauses than* SSR.

Proof. Consider the formula $F = (a \vee b) \wedge (\bar{b} \vee c) \wedge (a \vee \bar{c} \vee d)$. Since $\mathrm{HL}(F, a) = \{\bar{b}, \bar{c}\}$, HLE can remove literal $\bar{c}$ from the last clause in contrast to SSR. □

HLE can also strengthen formulas by increasing possibilities for unit propagation.

Proposition 4. *Removal of hidden literals can increase* BCP.

Proof. Consider the formula $F = (a \vee b) \wedge (\bar{b} \vee c) \wedge (a \vee \bar{c} \vee d)$. Since $\mathrm{HL}(F, a) = \{\bar{b}, \bar{c}\}$, HLE removes literal $\bar{c}$ from the last clause. When d is assigned to 0 after eliminating literal $\bar{c}$, BCP will infer a. □

In general, HLE does not have a unique fixpoint.

Proposition 5. *Applying* HLE *until fixpoint is not confluent.*

Proof. Consider the formula $F = (a \vee b) \wedge (\bar{a} \vee \bar{b}) \wedge (a \vee \bar{b} \vee c)$. Since $\mathrm{HL}(F, a) = \{\bar{b}\}$ and $\mathrm{HL}(F, \bar{b}) = \{a\}$, HLE can remove either $\bar{b}$ or a from $(a \vee \bar{b} \vee c)$. A fixpoint is reached after removing one of these two literals. □

In the example the non-confluence is due to a and $\bar{b}$ being equivalent literals. In fact, assume that all clauses in F_2 are kept even in the case HLE turns a binary clause into a unit clause (i.e., in such cases HLE will introduce new unit clauses into F). Then HLE can be made confluent (modulo variable renaming) by substituting equivalent literals.

Theorem 1. *For any CNF formula F, assuming that all clauses in the original F_2 are kept, alternating* ELS *and* HLE *(until fixpoint) until fixpoint is confluent modulo variable renaming.*

Proof sketch. ELS is confluent modulo variable renaming. Now consider HLE. Assume that we do not change F_2. Take any clause C with $l, l' \in C$ and $l' \in \mathrm{HL}(F, l)$. The only possible source of non-confluence is that $l \in \mathrm{HL}(F, l')$. Then there is a cycle in F_2, and hence l and $\bar{l}'$ are equivalent literals. This is handled by ELS afterwards. Now assume a binary clause is added to F_2 by HLE shortening a clause of length > 2. Newly produced cycles are handled by ELS afterwards. □

4 Unhiding Redundancies based on Time Stamping

In this section we present an efficient algorithm for detecting several kinds of redundancies in CNF formulas, focusing on techniques which exploit binary clauses.

For a given CNF formula F, our algorithm, referred to as *Unhiding* (see Fig. 1, details explained in the following), consists in essence of two phases. First, a depth-first search (DFS) over the binary implication graph $\mathsf{BIG}(F)$ is performed. During the DFS, each literal in $\mathsf{BIG}(F)$ is assigned a time stamp; we call this process *time stamping*. In the second phase, these time stamps are used for discovering the various kinds of redundancies in F, which are then removed.

In the following, we will first describe a *basic time stamping procedure* (Sect. 4.1). Then we will show how redundancies can be detected and eliminated based on the time stamps (Sect. 5). After these, in Sect. 6 we describe a more *advanced time stamping procedure* that embeds additional simplifications that are captured *during* the actual depth-first traversal of $\mathsf{BIG}(F)$.

4.1 Basic Time Stamping

The basic time stamping procedure implements a depth-first search on the binary implication graph $\mathsf{BIG}(F)$ of a given CNF formula F. The procedure associates a *discovered-finished interval* (or a *time stamp*) with each literal in $\mathsf{BIG}(F)$ according to the depth-first traversal order. For any depth-first traversal of a graph G, a node in G is *discovered* (resp. *finished*) the first (resp. last) time it is encountered during search. For a given depth-first traversal, the *discovery* and *finish times* of a node v in G, denoted by $\mathrm{dsc}(v)$ and $\mathrm{fin}(v)$, respectively, are defined as the number of steps taken at the time of discovering and finishing, respectively, the node v. The important observation here is that, according to the well-known "parenthesis theorem", for two nodes u and v with discovered-finished intervals $[\mathrm{dsc}(u), \mathrm{fin}(u)]$ and $[\mathrm{dsc}(v), \mathrm{fin}(v)]$, respectively, we

know that v is a descendant of u in the DFS tree if and only if $\mathrm{dsc}(u) < \mathrm{dsc}(v)$ and $\mathrm{fin}(u) > \mathrm{fin}(v)$, i.e., if the time stamp (interval) of u *contains* the time stamp (interval) of v. These conditions can be checked in constant time given the time stamps.

Pseudo-code for the main unhiding procedure *Unhiding* and the time stamping procedure *Stamp* is presented in Fig. 1. The main procedure *Unhiding* (left) initializes the attributes and calls the recursive stamping procedure (right) for each root in $\mathsf{BIG}(F)$ in a random order. When there are no more roots, we pick a literal not visited yet as the next starting point until all literals have been visited.[4] *Stamp* performs a DFS in $\mathsf{BIG}(F)$ from the given starting literal, assigns for each literal l encountered the discovery and finish times $\mathrm{dsc}(l)$ and $\mathrm{fin}(l)$ according to the traversal order, updates *stamp* (initially 0), and for each literal l, defines its DFS parent $\mathrm{prt}(l)$ and the root $\mathrm{root}(l)$ of the DFS tree in which l was discovered.

In the following, we say that a given time stamping *represents the implication* $l \rightarrow l'$ if the time stamp of l contains the time stamp of l'.

Unhiding (formula F)

1. $stamp := 0$
2. **foreach** literal l in $\mathsf{BIG}(F)$ **do**
3. $\mathrm{dsc}(l) := 0$; $\mathrm{fin}(l) := 0$
4. $\mathrm{prt}(l) := l$; $\mathrm{root}(l) := l$
5. **foreach** $r \in \mathrm{RTS}(F)$ **do**
6. $stamp :=$ *Stamp*$(r, stamp)$
7. **foreach** literal l in $\mathsf{BIG}(F)$ **do**
8.   **if** $\mathrm{dsc}(l) = 0$ **then**
9. $stamp :=$ *Stamp*$(l, stamp)$
10. **return** *Simplify*(F)

Stamp (literal l, integer $stamp$)

1. $stamp := stamp + 1$
2. $\mathrm{dsc}(l) := stamp$
3. **foreach** $(\bar{l} \vee l') \in F_2$ **do**
4.   **if** $\mathrm{dsc}(l') = 0$ **then**
5. $\mathrm{prt}(l') := l$
6. $\mathrm{root}(l') := \mathrm{root}(l)$
7. $stamp :=$ *Stamp*$(l', stamp)$
8. $stamp := stamp + 1$
9. $\mathrm{fin}(l) := stamp$
10. **return** $stamp$

Fig. 1. The *Unhiding* algorithm. Left: the main procedure. Right: the basic stamping procedure.

Example 1. Consider the formula

$$E = (\bar{a} \vee c) \wedge (\bar{a} \vee d) \wedge (\bar{b} \vee d) \wedge (\bar{b} \vee e) \wedge (\bar{c} \vee f) \wedge (\bar{d} \vee f) \wedge (\bar{f} \vee h) \wedge (\bar{g} \vee f) \wedge (\bar{g} \vee h) \wedge (\bar{a} \vee \bar{e} \vee h) \wedge (\bar{b} \vee \bar{c} \vee h) \wedge (a \vee b \vee c \vee d \vee e \vee f \vee g \vee h).$$

The formula contains several redundant clauses and literals. The clauses $(\bar{a} \vee \bar{e} \vee h)$, $(\bar{g} \vee h)$, and $(\bar{b} \vee \bar{c} \vee h)$ are hidden tautologies. In the last clause, all literals except e and h are hidden. The binary implication graph $\mathsf{BIG}(E)$ of E, as shown in Fig. 2, consists of two components. A partition of $\mathsf{BIG}(E)$ produced by the basic time stamping procedure is shown in Fig. 3. The nodes are visited in the following order: g, f, h, $\bar{e}$, $\bar{b}$, b, e, d, $\bar{h}$, $\bar{g}$, $\bar{f}$, $\bar{d}$, $\bar{a}$, $\bar{c}$, a, c. $\mathsf{BIG}(E)$ consists of 30 implications including the transitive ones. However, the trees and time stamps in the figure explicitly represent only 16 of them, again including transitive edges such as $\bar{h} \rightarrow \bar{a}$. The implications $b \rightarrow f$, $\bar{f} \rightarrow \bar{b}$, $b \rightarrow h$, and $\bar{h} \rightarrow \bar{b}$ are not represented by this time stamping. Note that the implication $\bar{f} \rightarrow \bar{c}$ is represented, and thus implicitly $c \rightarrow f$ as well. Using contraposition this way the four transitive edges mentioned above are not represented, the other 26 edges are.

[4] Thus, BIG needs not to be acyclic. Note that eliminating cycles in BIG by substituting variables might shorten longer clauses to binary clauses, which in turn could introduce new cycles. This process cannot be bounded to be linear and is not necessary for our algorithms.

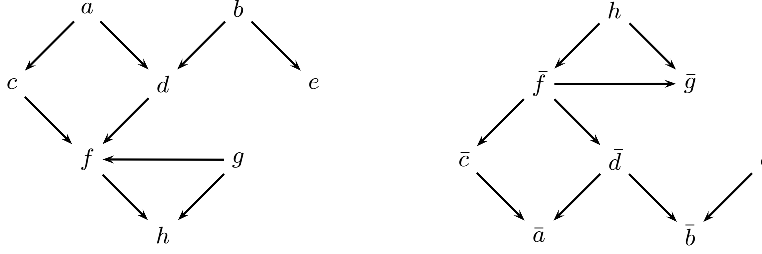

Fig. 2. $\mathsf{BIG}(E)$. The graph has five root nodes: a, b, $\bar{e}$, g, and $\bar{h}$.

The order in which the trees are traversed has a big impact on the quality, i.e. the fraction of implications that are represented by the time stamps. The example shows that randomized stamping may not represent all implications in BIG. Yet, for this formula, there is a DFS order that produces a stamping that represents *all* implications: start from the root $\bar{h}$ and stamp the tree starting with literal $\bar{f}$. Then, by selecting a as the root of the second tree, regardless of the order of the other roots and literals, the time stamps produced by stamping will represent all implications. ■

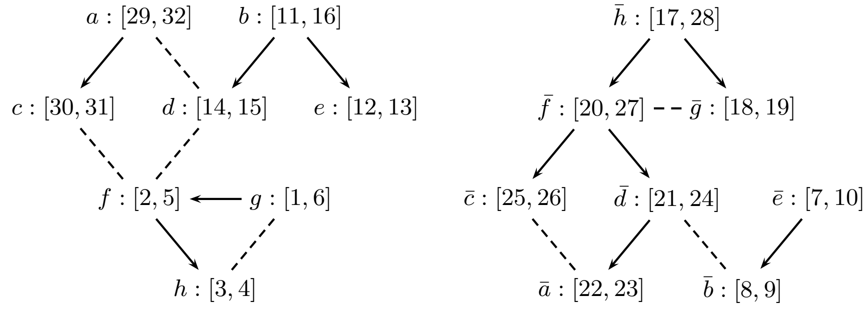

Fig. 3. A partition of $\mathsf{BIG}(E)$ into a forest with discovered-finished intervals $[\mathrm{dsc}(v), \mathrm{fin}(v)]$ assigned by the basic time stamping routine. Dashed lines represent implications in $\mathsf{BIG}(E)$ which are not used to set the time stamps.

5 Capturing Various Simplifications

We now explain how one can remove hidden literals and hidden tautologies, and furthermore perform hyper binary resolution steps based on a forest over the time stamped literal nodes produced by the main DFS procedure. The main procedure *Simplify* for this second phase, called by the main *Unhiding* procedure after time stamping, is shown in Fig. 4. For each clause C in the input CNF formula F, *Simplify* removes C from F. Then, it first checks whether the *UHTE* procedure detects that C is a hidden tautology. If not, literals are (possibly) eliminated from C by the *UHLE* procedure (using hidden literal elimination). The resulting clause is added to F.

Notice that the simplification procedure visits each clause $C \in F$ only once. The invoked sub-procedures, *UHTE* and *UHLE*, exploit the time stamps, and use two sorted lists: (i) $S^+(C)$, list of the literals in C sorted according to increasing discovery time, and (ii) $S^-(C)$, list of the complements of the literals in C, sorted according to increasing discovery time. We will now explain both of these sub-procedures in detail.

```
Simplify (formula F)
1   foreach C ∈ F
2       F := F \ {C}
3       if UHTE(C) then continue
4       F := F ∪ {UHLE(C)}
5   return F
```

Fig. 4. Procedure for applying HTE and HLE based on time stamps.

5.1 Hidden Literals

Once literals are stamped using the unhiding algorithm, one can cheaply detect (possibly a subset of) hidden literals. In this context, literal $l \in C$ is hidden if there is (i) an implication $l \to l'$ with $l' \in C$ that is represented by the time stamping, or (ii) an implication $\bar{l}' \to \bar{l}$ with $l' \in C$ that is represented by the time stamping.

We check for such implications as follows using the *UHLE* procedure shown in Fig. 5. For each input clause C, the procedure returns a subset of C with some hidden literals removed from C. For this procedure, we use $S^+(C)$ in reverse order, denoted by $S^+_{\text{rev}}(C)$. In essence, we go through the lists $S^+_{\text{rev}}(C)$ and $S^-(C)$, and compare the finish times of two successive elements in the lists. In case an implication is found, a hidden literal is detected and removed.

Lines 1-4 in Fig. 5 detect implications of the form $l \to l'$ with $l, l' \in C$ that are represented by the time stamping. Recall that in $S^+_{\text{rev}}(C)$ literals are ordered with decreasing discovering time. Let l' be located before l in $S^+_{\text{rev}}(C)$. If $\text{fin}(l) > \text{fin}(l')$ we found the implication $l \to l'$, and hence l is a hidden literal (in the code $\textit{finished} = \text{fin}(l')$). Line 3 checks whether the next element in $S^+_{\text{rev}}(C)$ is a hidden literal, and if so, the literal is removed. Lines 5-8 detect implications $\bar{l}' \to \bar{l}$ with $l, l' \in C$. In $S^-(C)$ literals are ordered with increasing discovering time. Now, $\bar{l}'$ be located before $\bar{l}$ in $S^-(C)$ and $\textit{finished} = \text{fin}(\bar{l}')$. On Line 7 we check that $\text{fin}(\bar{l}) < \text{fin}(\bar{l}')$ or, equivalently, $\text{fin}(\bar{l}) < \textit{finished}$. In that case l is a hidden literal and is hence removed.

Example 2. Recall the formula E from Example 1. All literals except e and h in the clause $C = (a \vee b \vee c \vee d \vee e \vee f \vee g \vee h) \in E$ are hidden. In case the literals in $\text{RTS}(E)$ are stamped with the time stamps shown in Figure 3, the *UHLE* procedure can detect them all. Consider first the sequence $S^+_{\text{rev}}(C) = (c, a, d, e, b, h, f, g)$. Since $\text{fin}(c) < \text{fin}(a)$, a is removed from C. Similarly, $\text{fin}(e) < \text{fin}(b)$ and $\text{fin}(f) < \text{fin}(g)$, and hence b and g are removed from C. Second, consider the complements of the literals in the reduced clause: $S^-(C) = (\bar{e}, \bar{h}, \bar{f}, \bar{d}, \bar{c})$. Now, $\text{fin}(\bar{h}) > \text{fin}(\bar{f}), \text{fin}(\bar{d}), \text{fin}(\bar{c})$, and hence f, d, and c are removed. ■

```
UHLE (clause C)
1   finished := finish time of first element in S+rev(C)
2   foreach l ∈ S+rev(C) starting at second element
3       if fin(l) > finished then C := C \ {l}
4       else finished := fin(l)
5   finished := finish time of first element in S−(C)
6   foreach l̄ ∈ S−(C) starting at second element
7       if fin(l̄) < finished then C := C \ {l}
8       else finished := fin(l̄)
9   return C
```

Fig. 5. Eliminating hidden literals using time stamps.

5.2 Hidden Tautologies

Fig. 6 shows the pseudo-code for the *UHTE* procedure that detects hidden tautologies based on time stamps. Notice that if a time stamping represents an implication of the form $\bar{l} \rightarrow l'$, where both l and l' occur in a clause C, then the clause C is a hidden tautology.

The *UHTE* procedure goes through the sorted lists $S^+(C)$ and $S^-(C)$ to find two literals $l_{\text{neg}} \in S^-(C)$ and $l_{\text{pos}} \in S^+(C)$ such that the time stamping represents the implication $l_{\text{neg}} \rightarrow l_{\text{pos}}$, i.e., it checks if $\text{dsc}(l_{\text{neg}}) < \text{dsc}(l_{\text{pos}})$ and $\text{fin}(l_{\text{neg}}) > \text{fin}(l_{\text{pos}})$. The procedure starts with the first literals $l_{\text{neg}} \in S^-(C)$ and $l_{\text{pos}} \in S^+(C)$, and loops through the literals in $l_{\text{pos}} \in S^+(C)$ until $\text{dsc}(l_{\text{neg}}) < \text{dsc}(l_{\text{pos}})$ (Lines 4–6). Once such a l_{pos} is found, if $\text{fin}(l_{\text{neg}}) > \text{fin}(l_{\text{pos}})$ (Line 7), we know that C is a hidden tautology, and the procedure returns true (Line 10). Otherwise, we loop through $S^-(C)$ to select a new l_{neg} for which the condition holds (Lines 7–9). Then (Lines 4–6), if $\text{dsc}(l_{\text{neg}}) < \text{dsc}(l_{\text{pos}})$, C is a hidden tautology. Otherwise, we select a new l_{pos}. Unless a hidden tautology is detected, the procedure terminates once it has looped through all literals in either $S^+(C)$ or $S^-(C)$ (Lines 5 and 8).

One has to be careful while removing binary clauses based on time stamps. There are two exceptions in which time stamping represents an implication $l_{\text{neg}} \rightarrow l_{\text{pos}}$ with $l_{\text{neg}} \in S^-(C)$ and $l_{\text{pos}} \in S^+(C)$ for which C is not a hidden tautology. First, if $l_{\text{pos}} = \bar{l}_{\text{neg}}$, then l_{neg} is a failed literal. Second, if $\text{prt}(l_{\text{pos}}) = l_{\text{neg}}$, then C was used to set the time stamp of l_{pos}. Line 7 takes both of these cases into account.

Example 3. Recall again the formula E from Example 1. E contains three hidden tautologies: $(\bar{g} \vee h)$, $(\bar{a} \vee \bar{e} \vee h)$, and $(\bar{b} \vee \bar{c} \vee h)$. In the time stamping in Fig. 3, $\bar{h} : [17, 28]$ contains $\bar{g} : [18, 19]$. However, $\text{prt}(\bar{g}) = \bar{h}$, and hence $(\bar{g} \vee h)$ cannot be removed. On the other hand, $\bar{g} : [1, 6]$ contains $\bar{h} : [3, 4]$, and $\text{prt}(h) \neq g$, and hence $(\bar{g} \vee h)$ is identified as a hidden tautology. We can also identify $(\bar{a} \vee \bar{e} \vee h)$ as a hidden tautology because $\bar{h} : [17, 28]$ contains $\bar{a} : [22, 23]$. This is not the case for $(\bar{b} \vee \bar{c} \vee h)$ because the implications $b \rightarrow h$ and $\bar{h} \rightarrow \bar{b}$ are not represented by the time stamping. ■

Proposition 6. *For any Unhiding time stamping, UHTE detects all hidden tautologies that are represented by the time stamping.*

Proof sketch. For every $l_{\text{neg}} \in S^-(C)$, *UHTE* checks if time stamping represents the implication $l_{\text{neg}} \rightarrow l_{\text{pos}}$ for the first literal in $l_{\text{pos}} \in S^+(C)$ for which $\text{dsc}(l_{\text{neg}}) <$

```
UHTE (clause C)
1    l_pos := first element in S+(C)
2    l_neg := first element in S−(C)
3    while true
4      if dsc(l_neg) > dsc(l_pos) then
5        if l_pos is last element in S+(C) then return false
6        l_pos := next element in S+(C)
7      else if fin(l_neg) < fin(l_pos) or (|C| = 2 and (l_pos = ¬l_neg or prt(l_pos) = l_neg)) then
8        if l_neg is last element in S−(C) then return false
9        l_neg := next element in S−(C)
10     else return true
```

Fig. 6. Detecting hidden tautologies using time stamps.

$\mathrm{dsc}(l_{\mathrm{pos}})$ holds. The key observation is that if there is a $l_{\mathrm{neg}} \in S^{-}(C)$ and a $l_{\mathrm{pos}} \in S^{+}(C)$ such that time stamping represents the implication $l_{\mathrm{neg}} \rightarrow l_{\mathrm{pos}}$, then the stamps also represent $l_{\mathrm{neg}} \rightarrow l'_{\mathrm{pos}}$ with l'_{pos} being the first literal in $S^{+}(C)$ for which $\mathrm{dsc}(l_{\mathrm{neg}}) < \mathrm{dsc}(l_{\mathrm{pos}})$ holds. □

If a clause C is a hidden tautology, then $\mathrm{HLA}(F, C)$ is a hidden tautology due to $\mathrm{HLA}(F, C) \supseteq C$. However, it is possible that, for a given clause C, *UHTE*(C) returns true, while *UHTE*(*UHLE*(C)) returns false. In other words, *UHLE* could in some cases disrupt *UHTE*. For instance, consider the clause $(a \vee b \vee c)$ and the following time stamps: $a : [2, 3]$, $\bar{a} : [9, 10]$, $b : [1, 4]$, $\bar{b} : [5, 8]$, $c : [6, 7]$, $\bar{c} : [11, 12]$. Now *UHLE* removes literal b which is required for *UHTE* to return true. Therefore *UHTE* should be called before *UHLE*, as is done in our *Simplify* procedure (recall Fig. 4).

5.3 Adding Hyper Binary Resolution

An additional binary clause based simplification technique that can be integrated into the unhiding procedure is *hyper binary resolution* [1] (HBR). Given a clause of the form $(l_1 \vee \cdots \vee l_k)$ and $k - 1$ binary clauses of the form $(l' \vee \bar{l}_i)$, where $2 \leq i \leq k$, the hyper binary resolution rule allows to infer the clause $(l_1 \vee l')$ in one step.

For HBR in the unhiding algorithm we only need the list $S^{-}(C)$. Let C be a clause with k literals. We find a hyper binary resolvent if (i) all literals in $S^{-}(C)$, except the first one l_1, have a common ancestor l', or (ii) all literals in $S^{-}(C)$, except the last one l_k, have a common ancestor l''. In case (i) we find $(l_1 \vee \bar{l}')$, and in case (ii) we find $(l_k \vee \bar{l}'')$. It is even possible that all literals in $S^{-}(C)$ have a common ancestor l''' which shows that l''' is a failed literal, in which case we can learn the unit clause $(\bar{l}''')$.

While *UHBR*(C) could be called in *Simplify* after Line 4, our experiments show that applying *UHBR*(C) does not give further gains w.r.t. running times, and can in cases degrade performance. We suspect that this is because *UHBR*(C) may add transitive edges to $\mathsf{BIG}(F)$. Consider the formula $F = (a \vee b \vee c) \wedge (\bar{a} \vee d) \wedge (\bar{b} \vee d) \wedge (c \vee e) \wedge (c \vee f) \wedge (d \vee \bar{e})$. Assume that the time stamping DFS visits the literals in the order $\bar{f}$, c, a, d, $\bar{d}$, $\bar{e}$, $\bar{a}$, $\bar{b}$, $\bar{c}$, f, e, b. *UHBR*($(a \vee b \vee c)$) can learn $(c \vee d)$, but it cannot check that this binary clause adds a transitive edge to $\mathsf{BIG}(F)$.

5.4 Some Limitations of Basic Stamping

As already pointed out, time stamps produced by randomized DFS may not represent all implications of F_2. In fact, the fraction of implications represented can be very small in the worst case. Especially, consider the formula $F = (a \vee b \vee c \vee d) \wedge (\bar{a} \vee \bar{b}) \wedge (\bar{a} \vee \bar{c}) \wedge (\bar{a} \vee \bar{d}) \wedge (\bar{b} \vee \bar{c}) \wedge (\bar{b} \vee \bar{d}) \wedge (\bar{c} \vee \bar{d})$ that encodes that exactly one of a, b, c, d must be true. Due to symmetry, there is only one possible DFS traversal order, and it produces the time stamps $a : [1, 8], \bar{b} : [2, 3], \bar{c} : [4, 5], \bar{d} : [6, 7], b : [9, 12]$, $\bar{a} : [10, 11], c : [13, 14], d : [15, 16]$. Only three of the six binary clauses are represented by the time stamps. This example can be extended to n variables, in which case only $n - 1$ of the $n(n-1)/2$ binary clauses are represented. In order to capture as many implications (and thus simplification opportunities) as possible, in practice we apply multiple repetitions of *Unhiding* using randomized DFS (as detailed in Sect. 7).

6 Advanced Stamping for Capturing Additional Simplifications

In this section we develop an advanced version of the DFS time stamping procedure. Our algorithm can be seen as an extension of the BinSATSCC-1 algorithm in [13]. The advanced procedure, presented in Fig. 7, enables performing additional simplifications *on-the-fly during* the actual time stamping phase: the on-the-fly techniques can perform some simplifications that cannot be done with *Simplify*(F), and, on the other hand, enlarging the time stamps of literals may allow further simplifications in *Simplify*(F). Although not discussed further in this paper due to the page limit, we note that, additionally, all simplifications by *UHTE*, *UHLE*, and *UHBR* which only use binary clauses could be performed on-the-fly within the advanced stamping procedure.

Here we introduce the attribute $\mathrm{obs}(l)$ that denotes the latest time point of observing l. The value of $\mathrm{obs}(l)$ can change frequently during *Unhiding*. Each line of the advanced stamping procedure (Fig. 7) is labeled. The line labeled with `OBS` assigns $\mathrm{obs}(l)$ for literal l. The label `BSC` denotes that the line originates from the basic stamping procedure (Fig. 1). Lines with the other labels are techniques that can be performed on-the-fly: transitive reduction (`TRD` / Sect. 6.1), failed literal elimination (`FLE` / Sect. 6.2), and equivalent literal substitution (`ELS` / Sect. 6.3). The technique `TRD` depends on `FLE` and both techniques use the $\mathrm{obs}()$ attribute while `ELS` is independent of $\mathrm{obs}()$.

```
            Stamp (literal l, integer stamp)
 1 BSC        stamp := stamp + 1
 2 BSC/OBS    dsc(l) := stamp; obs(l) := stamp
 3 ELS        flag := true                                  // l represents a SCC
 4 ELS        S.push(l)                                     // push l on SCC stack
 5 BSC        for each (¬l ∨ l') ∈ F_2
 6 TRD          if dsc(l) < obs(l') then F := F \ {(¬l ∨ l')}; continue
 7 FLE          if dsc(root(l)) ≤ obs(¬l') then
 8 FLE            l_failed := l
 9 FLE            while dsc(l_failed) > obs(¬l') do l_failed := prt(l_failed)
10 FLE            F := F ∪ {(¬l_failed)}
11 FLE            if dsc(¬l') ≠ 0 and fin(¬l') = 0 then continue
12 BSC          if dsc(l') = 0 then
13 BSC            prt(l') := l
14 BSC            root(l') := root(l)
15 BSC            stamp := Stamp(l', stamp)
16 ELS          if fin(l') = 0 and dsc(l') < dsc(l) then
17 ELS            dsc(l) := dsc(l'); flag := false          // l is equivalent to l'
18 OBS          obs(l') := stamp                            // set last observed time attribute
19 ELS        if flag = true then                           // if l represents a SCC
20 BSC          stamp := stamp + 1
21 ELS          do
22 ELS            l' := S.pop()                             // get equivalent literal
23 ELS            dsc(l') := dsc(l)                         // assign equal discovered time
24 BSC            fin(l') := stamp                          // assign equal finished time
25 ELS          while l' ≠ l
26 BSC        return stamp
```

Fig. 7. Advanced literal time stamping capturing failed and equivalent literals

6.1 Transitive Reduction

Binary clauses that represent transitive edges in BIG are in fact hidden tautologies [6]. Such clauses can already be detected in the stamping phase (i.e., before *UHTE*), as shown in the advanced stamping procedure on Line 6 with label TRD.

A binary clause $(\bar{l} \vee l')$ can only be observed as a hidden tautology if $\mathrm{dsc}(l') > 0$ during *Stamp*$(l, stamp)$. Otherwise, $\mathrm{prt}(l') := l$, which satisfies the last condition on Line 7 of *UHTE*. If $\mathrm{dsc}(l') > \mathrm{dsc}(l)$ just before calling *Stamp*$(l', stamp)$, then $(\bar{l} \vee l')$ is a hidden tautology. When transitive edges are removed on-the-fly, *UHTE* can focus on clauses of size ≥ 3, making the last check on Line 7 of *UHTE* redundant.

Transitive edges in $\mathsf{BIG}(F)$ can hinder the unhiding algorithm by reducing the time stamp intervals. Hence as many transitive edges as possible should be removed. Notice that in case $0 < \mathrm{dsc}(l') < \mathrm{dsc}(l)$, *Stamp*$(l, stamp)$ cannot detect that $(\bar{l} \vee l')$ is a hidden tautology. Yet by using $\mathrm{obs}(l')$ instead of $\mathrm{dsc}(l')$ in the check (Line 14 of Fig. 7), we can detect additional transitive edges. For instance, consider the formula $F = (\bar{a} \vee b) \wedge (b \vee \bar{c}) \wedge (b \vee \bar{d}) \wedge (\bar{c} \vee d)$ where $(b \vee \bar{c})$ is a hidden tautology. If *Unhiding* visits the literals in the order $a, b, c, d, \bar{b}, \bar{a}, \bar{c}, \bar{d}$, then this hidden tautology is not detected using $\mathrm{dsc}(l')$. However, while visiting d in advanced stamping, we assign $\mathrm{obs}(b) := \mathrm{dsc}(d)$. Now, using $\mathrm{obs}(l')$, *Stamp*$(c, stamp)$ can detect that $(b \vee \bar{c})$ is a hidden tautology.

6.2 Failed Literal Elimination over F_2

Detection of failed literals in F_2 can be performed on-the-fly during stamping. If a literal l in F_2 is failed, then all ancestors of l in $\mathsf{BIG}(F)$ are also failed. Recall that there is a strong relation between HLE restricted to F_2 and failed literals in F_2 (Prop. 1).

To detect a failed literal, we check for each observed literal l' whether $\bar{l}'$ was also observed in the current tree, or $\mathrm{dsc}(\mathrm{root}(l)) \leq \mathrm{dsc}(\bar{l}')$. In that case the lowest common ancestor in the current tree is a failed literal. Similar to transitive reduction, the number of detected failed literals can be increased by using the $\mathrm{obs}(\bar{l}')$ attribute instead of $\mathrm{dsc}(\bar{l}')$. We compute the lowest common ancestor l_{failed} of l' and $\bar{l}'$ (Lines 8–9 in Fig. 7). Afterwards the unit clause $(\bar{l}_{\mathrm{failed}})$ is added to the formula (Line 10).

At the end of on-the-fly FLE (Line 11), the advanced stamping procedure checks whether to stamp l' after finding a failed literal. In case we learned that $\bar{l}'$ is a failed literal, then we have the unit clause (l'). Then it does not make sense to stamp l', as all implications of l' can be assigned to true by BCP. This check also ensures that binary clauses currently used in the recursion are not removed by transitive reduction.

6.3 Equivalent Literal Substitution

In case $\mathsf{BIG}(F)$ contains a cycle, then all literals in that cycle are equivalent. In the basic stamping procedure all these literals will be assigned a different time stamp. Therefore, many implications of F_2 will not be represented by any of the resulting time stampings. To fix this problem, equivalent literals should be assigned the same time stamps.

A cycle in $\mathsf{BIG}(F)$ can be detected after calling *Stamp*$(l', stamp)$, by checking whether $\mathrm{fin}(l')$ still has the initial value 0. This check can only return true if l' is an ancestor of l. We implemented ELS on-the-fly using a variant of Tarjan's SCC decomposition algorithm [17] which detects all cycles in $\mathsf{BIG}(F)$ using any depth-first traversal order. We use a local boolean *flag* that is initialized to true (Line 3). If true,

$flag$ denotes that l represents a SCC. In case it detects a cycle, $flag$ is set to false (Lines 16–17). Additionally, a global stack S of literals is used, and is initially empty. At each call of *Stamp*$(l, stamp)$, l is pushed on the stack (Line 4). At the end of the procedure, if l is still the representative of a SCC, all literals in S being equivalent to l, all literals in S are assigned the same time stamp (Lines 19–25).

7 Experiments

We have implemented *Unhiding* in our state-of-the-art SAT solver Lingeling [12] (version 517, source code and experimental data at http://fmv.jku.at/unhiding) as an additional preprocessing or, more precisely, *inprocessing* technique applied during search. Batches of randomized unhiding rounds are interleaved with search and other already included inprocessing techniques. The number of unhiding rounds per unhiding phase and the overall work spent in unhiding is limited in a similar way as is already done in Lingeling for the other inprocessing. The cost of *Unhiding* is measured in the number of recursive calls to the stamping procedure and the number of clauses traversed. Sorting clauses (in *UHTE* and *UHLE*) incurs an additional penalty. In the experiments *Unhiding* takes on average roughly 7% of the total running time (including search), which is more than twice as much as standard failed literal probing (2%) and around half of the time spent on SatElite-style variable elimination (16%).

The cluster machines used for the experiments, with Intel Core 2 Duo Quad Q9550 2.8-GHz processors, 8-GB main memory, running Ubuntu Linux version 9.04, are around twice as fast as the ones used in the first phase of the 2009 SAT competition. For the experiments we used a 900 s timeout and a memory limit of 7 GB. Using the set of all 292 application instances from SAT Competition 2009 (http://satcompetition.org/2009/), a comparison of the number of solved instances for different configurations of *Unhiding* and the baseline (up-to-date version of Lingeling without *Unhiding*) is presented in Table 1. Note that we obtained similar results also for the SAT Race 2010 instances, and also improved performance on the crafted instances of SAT Competition 2009.

Table 1. Comparison of different configurations of *Unhiding* and the baseline solver Lingeling. The 2nd to 4th columns show the number of solved instances (sol), resp. solved satisfiable (sat) and unsatisfiable (uns) instances. The next three columns contain the average percentage of total time spent in unhiding (unhd), all simplifications through inprocessing (simp), and variable elimination (elim). Here we also take unsolved instances into account. The rest of the table lists the number of hidden tautologies (hte) in millions, the number of hidden literal eliminations (hle), also in millions, and finally the number of unhidden units (unts) in thousands which includes the number of unhidden failed literals. We also include the average percentage (stp) of hidden tautologies resp. derived units during stamping, and the average percentage (red) of redundant/learned hidden tautologies resp. removed literals in redundant/learned clauses. A more detailed analysis shows that for many instances, the percentage of redundant clauses is very high, actually close to 100%, both for HTE and HLE. Note that "unts" is not precise as the same failed literal might be found several times during stamping since we propagate units lazily after unhiding.

configuration	sol	sat	uns	unhd	simp	elim	hte	stp	red	hle	red	unts	stp
adv.stamp (no uhbr)	188	78	110	7.1%	33.0%	16.1%	22	64%	59%	291	77.6%	935	57%
adv.stamp (w/uhbr)	184	75	109	7.6%	32.8%	15.8%	26	67%	70%	278	77.9%	941	58%
basic stamp (no uhbr)	183	73	110	6.8%	32.3%	15.8%	6	0%	52%	296	78.0%	273	0%
basic stamp (w/uhbr)	183	73	110	7.4%	32.8%	15.8%	7	0%	66%	288	76.7%	308	0%
no unhiding	180	74	106	0.0%	28.6%	17.6%	0	0%	0%	0	0.0%	0	0%

The three main observations are: (i) *Unhiding* increases the number of solved satisfiable instances already when using the basic stamping procedure; (ii) using the advanced stamping scheme, the number of solved instances increases notably for both satisfiable and unsatisfiable instances; and (iii) the *UHBR* procedure actually degrades the performance (in-line with the discussion in Sect. 5.3). Hence the main advantages of *Unhiding* are due to the combination of the advanced stamping procedure, *UHTE*, and *UHLE*.

8 Conclusions

The *Unhiding* algorithm efficiently (close to linear time) approximates a combination of binary clause based simplifications that is conjectured to be at least quadratic in the worst case. In addition to applying known simplification techniques, including the recent hidden tautology elimination, we introduced the novel technique of hidden literal elimination, and implemented it within *Unhiding*. We showed that *Unhiding* improves the performance of a state-of-the-art CDCL SAT solver when integrated into the search procedure for inprocessing formulas (including learnt clauses) during search.

References

1. Bacchus, F.: Enhancing Davis Putnam with extended binary clause reasoning. In: Proc. AAAI, AAAI Press (2002) 613–619
2. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Proc. SAT. Volume 3569 of LNCS., Springer (2005) 61–75
3. Gershman, R., Strichman, O.: Cost-effective hyper-resolution for preprocessing CNF formulas. In: Proc. SAT. Volume 3569 of LNCS., Springer (2005) 423–429
4. Han, H., Somenzi, F.: Alembic: An efficient algorithm for CNF preprocessing. In: Proc. DAC, IEEE (2007) 582–587
5. Järvisalo, M., Biere, A., M. Heule, M.J.H.: Blocked clause elimination. In: Proc. TACAS. Volume 6015 of LNCS., Springer (2010) 129–144
6. Heule, M.J.H., Järvisalo, M., Biere, A.: Clause elimination procedures for CNF formulas. In: Proc. LPAR-17. Volume 6397 of LNCS., Springer (2010) 357–371
7. Marques Silva, J.P.: Algebraic simplification techniques for propositional satisfiability. In: Proc. CP. Volume 1894 of LNCS., Springer (2000) 537–542
8. Van Gelder, A.: Toward leaner binary-clause reasoning in a satisfiability solver. Annals of Mathematics and Artificial Intelligence **43**(1) (2005) 239–253
9. Li, C.M.: Integrating equivalency reasoning into Davis-Putnam procedure. In: Proc. AAAI. (2000) 291–296
10. Brafman, R.: A simplifier for propositional formulas with many binary clauses. IEEE Transactions on Systems, Man, and Cybernetics, Part B **34**(1) (2004) 52–59
11. Aho, A., Garey, M., Ullman, J.: The transitive reduction of a directed graph. SIAM Journal on Computing **1**(2) (1972) 131–137
12. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. FMV Report Series Technical Report 10/1, Johannes Kepler University, Linz, Austria (2010)
13. del Val, A.: Simplifying binary propositional theories into connected components twice as fast. In: Proc. LPAR. Volume 2250 of LNCS., Springer (2001) 392–406
14. Soos, M.: Cryptominisat 2.5.0, sat race 2010 solver description (2010)
15. Korovin, K.: iProver - an instantiation-based theorem prover for first-order logic. In: Proc. IJCAR. Volume 5195 of LNCS., Springer (2008) 292–298
16. Groote, J.F., Warners, J.P.: The propositional formula checker HeerHugo. J. Autom. Reasoning **24**(1/2) (2000) 101–125
17. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Computing **1**(2) (1972)