Factoring Out Assumptions to Speed Up MUS Extraction*

Jean-Marie Lagniez and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

Abstract. In earlier work on a limited form of extended resolution for CDCL based SAT solving, new literals were introduced to factor out parts of learned clauses. The main goal was to shorten clauses, reduce proof size and memory usage and thus speed up propagation and conflict analysis. Even though some reduction was achieved, the effectiveness of this technique was rather modest for generic SAT solving. In this paper we show that factoring out literals is particularly useful for incremental SAT solving, based on assumptions. This is the most common approach for incremental SAT solving and was pioneered by the authors of MINISAT. Our first contribution is to focus on factoring out only assumptions, and actually all eagerly. This enables the use of compact dedicated data structures, and naturally suggests a new form of clause minimization, our second contribution. As last main contribution, we propose to use these data structures to maintain a partial proof trace for learned clauses with assumptions, which gives us a cheap way to flush useless learned clauses. In order to evaluate the effectiveness of our techniques we implemented them within the version of MINISAT used in the publically available state-of-the-art MUS extractor MUSer. An extensive experimental evaluation shows that factoring out assumptions in combination with our novel clause minimization procedure and eager clause removal is particularly effective in reducing average clause size, improves running time and in general the state-of-the-art in MUS extraction.

1 Introduction

The currently most widespread approach for *incremental* SAT was pioneered by the authors of MINISAT [1] in context of bounded model checking [2] and finite model finding [3], and has seen many other important practical applications since then. It can easily be implemented on top of a standard SAT solver based on the *conflict driven clause learning* (CDCL) paradigm [4], as described in [1], by modifying the heuristics for picking decisions, to branch on literals assumed to be true first. In this paper we refer with *assumptions* to this set of literals assumed to be true.

Another important application, which makes use of incremental SAT, is the extraction of a *minimal unsatisfiable set* (MUS) of clauses from a propositional

* Supported by FWF, NFN Grant S11408-N23 (RiSE).

formula in *conjunctive normal form* (CNF). The current state-of-the-art in MUS extraction [5] is based on incremental SAT. In the context of MUS extraction [6,7,8,9,10], the focus of this paper, and in similar or related applications of incremental SAT [3,11,12,13,14,15,16], an additional analysis is required, which learns sub-sets of assumptions, under which the formula is proven to be unsatisfiable. In these applications, the number of assumptions is usually not only quite large, e.g. similar in size to the number of original variables and clauses in the CNF, but also the SAT solver is called many times, while the set of assumptions almost stays the same.

As it turns out, current SAT solvers have not been optimized for this actually rather common usage scenario. We propose a new technique for compressing incremental proofs for problems with many assumptions. Our technique is based on the idea of factoring out literals of learned clauses by extended resolution steps, which also forms the basis of related work on speeding up SAT solving in general [17,18]. Clauses *learned* in those applications we are interested in typically contain many literals which are the negation of original assumptions. We call these negations of originally assumed literals also assumptions or more precisely *assumption literals*, if the context requires to distinguish between originally assumed literals ("assumptions") used as decisions and their negations occurring in learned clauses ("assumption literals").

In our approach we factor out these assumption literals in order to shrink learned clauses and reduce the number of literals traversed, particularly during boolean constraint propagation (BCP). This idea, if implemented correctly, does not change the search at all, but it is still quite effective in reducing the time needed for MUS extraction. Further, factoring out assumptions enables the use of compact dedicated data structures, and naturally suggests a new form of clause minimization, which gives another substantial improvement. Recording factored out assumptions explicitly, also gives us simple way to maintain a partial proof trace for learned clauses with assumptions. The trace can be used to compute an approximation of a "clausal core". We can then discard learned clauses out-side this clausal core eagerly, which empirically seems to be a useful strategy.

The authors of [19] observed a similar deficiency when using the assumption based approach for incremental SAT solving in the context of bounded model checking. They propose to use an additional SAT solver, to which assumptions are added as unit clauses. This in turn allows to improve efficiency of preprocessing and inprocessing under assumptions, but prohibits to reuse in the main solver clauses learned by the additional solver. However, according to [19] it is possible, by selectively adding assumptions, to extract "pervasive clauses" from the resolution proofs of clauses learned in the additional solver, with the objective that adding these "pervasive clauses" to the main solver is sound.

While in [19] as in our approach some sort of resolution proof has to be maintained, the main solver in [19] still uses the classical assumption based approach and thus will benefit from our proposed techniques. Finally, the motivations as well as the application characteristics considered in the experimental part differ.

2 Factoring Out Assumptions

In incremental SAT with *many* assumptions, learned clauses contain many assumption literals too (see previous section for the definition of this terminology). Accordingly the average size of learned clauses can become very large (as we will see in Fig. 6). This effect increases the size of the working set (used memory), or more specifically, the average number of traversed literals per visited clause during BCP. The same argument applies to visited clauses during conflict analysis. As a consequence, SAT solver performance degrades.

For every learned clause we propose to replace the "assumption part" by a new fresh literal, called *abbreviation* literal. The replaced part consists of all assumptions and previously added abbreviations. The connection between the abbreviation and the replaced literals is stored in a *definition map* as follows.

$$\boxed{\begin{array}{c}(p_1 \vee \cdots \vee p_n \vee a_1 \vee \cdots \vee a_m)\\ \text{is factored out into}\\ (p_1 \vee \cdots \vee p_n \vee \ell) \qquad \text{and} \qquad \ell \mapsto \underbrace{a_1 \vee \cdots \vee a_m}_{\mathcal{G}[\ell]}\end{array}}$$

Fig. 1. Factoring out assumptions by introducing a new abbreviation literal ℓ.

Let $p_1 \vee \cdots \vee p_n \vee a_1 \vee \cdots \vee a_m$ be a new learned clause, where $p_1, \ldots, p_n$ are original literals and $a_1, \ldots, a_m$ are either assumptions or abbreviations. We pick a fresh abbreviation literal ℓ and instead of the originally learned clause add the clause $p_1 \vee \cdots \vee p_n \vee \ell$ to the clause data base. Then we record $a_1 \vee \cdots \vee a_m$ as the *definition* $\mathcal{G}[\ell]$ of ℓ in the definition map $\mathcal{G}$ (see Fig. 1). For $m \leq 1$ this replacement does not make sense and the original learned clause is kept instead.

Consider the example in Fig. 2 for an (incremental) SAT run under the assumptions $\overline{a_1}, \ldots, \overline{a_6}$. Conflict analysis might learn clauses α_1, ..., α_7 depicted on the left of Fig. 2(a), where $p_1, \ldots, p_7$ are original literals and $a_1, \ldots, a_6$ assumption literals.[1] Note that the run is not supposed to be complete. Only some clauses are shown together with their antecedent clauses, and original clauses are ignored too (to simplify the example). For instance α_3 is derived through resolution from α_1 and from some other original clauses not shown (the "...").

The result of introducing abbreviations to factor out assumptions is shown on the right. The first clause α_1 is factored into α'_1 and the definition $a_1 \vee a_2$ of the new abbreviation literal ℓ_1. The definition is recorded in the definition map, as shown in Fig. 2(b), where ℓ_1 has two incoming arcs, one from a_1 and one from a_2. Further let us point out, that $\alpha'_5 = \alpha_5$, because it *keeps* a_2 as single non-original literal, which (as discussed above) reduces the overall number of introduced abbreviations. Finally, note how definitions might recursively depend on other definitions as for ℓ_3, ℓ_4 or ℓ_5, while factoring α_3, α_4, and α_6 respectively.

As briefly discussed above, assumptions are always assigned first and thus assigning them can actually be seen as a preprocessing resp. initialization step

[1] Assumption literals are literals made of a variable which is currently used or was used in an assumption. See again the introduction section for a precise definition.

learned clauses	antecedents		factored clauses
$\alpha_1 : p_2 \vee p_7 \vee a_1 \vee a_2$	$\{\ldots\}$		$\alpha'_1 : p_2 \vee p_7 \vee \ell_1$
$\alpha_2 : p_2 \vee a_2 \vee a_3$	$\{\ldots\}$		$\alpha'_2 : p_2 \vee \ell_2$
$\alpha_3 : p_7 \vee p_4 \vee \overline{p_6} \vee a_1 \vee a_2 \vee a_4$	$\{\alpha_1, \ldots\}$	*factoring*	$\alpha'_3 : p_7 \vee p_4 \vee \overline{p_6} \vee \ell_3$
$\alpha_4 : p_6 \vee p_8 \vee a_3 \vee a_2 \vee a_5$	$\{\alpha_2, \ldots\}$	$\Longrightarrow$	$\alpha'_4 : p_6 \vee p_8 \vee \ell_4$
$\alpha_5 : p_2 \vee p_5 \vee a_2$	$\{\ldots\}$		$\alpha'_5 : p_2 \vee p_5 \vee a_2$
$\alpha_6 : p_7 \vee p_4 \vee a_1 \vee a_2 \vee a_4 \vee a_5$	$\{\alpha_3, \alpha_4, \ldots\}$		$\alpha'_6 : p_7 \vee p_4 \vee \ell_5$
$\alpha_7 : \overline{p_2} \vee a_6 \vee a_5$	$\{\ldots\}$		$\alpha'_7 : \overline{p_2} \vee \ell_6$

(a) Learned clauses (original version left, factored version right)

$$\begin{tikzcd}
a_6 \arrow[r] & \ell_6 \\
a_5 \arrow[ru] \arrow[rr] & & \ell_4 \arrow[ddddr] \\
a_3 \arrow[r] & \ell_2 \arrow[ru] \\
a_2 \arrow[ru] \arrow[r] & \ell_1 \arrow[ddr] \\
a_1 \arrow[ru] \\
a_4 \arrow[rr] & & \ell_3 \arrow[r] & \ell_5
\end{tikzcd}$$

(b) Definition Map

Fig. 2. Factoring out assumptions

before the actual solving starts. Furthermore, the algorithm for MUS extraction, as implemented in MUSer [9], to which we applied our technique, has the following property: *the set of variables used in assumptions stays the same over all incremental calls*, with the exception of variables assigned at the top-level. The techniques presented in this paper are sound, even if this property does not hold, i.e. the set of assumptions changes (substantially) from one incremental call to the next. However, if the property does not hold they are probably less effective. We focus on the important problem of MUS extraction here and leave it to future work to apply our techniques to other scenarios of incremental SAT.

Assigning in every incremental call the current set of assumptions during an initialization phase, will imply a unique value for all the (previously introduced) abbreviation literals, unless the set of assumptions turns out to be inconsistent, in which case the solver returns immediately. For that reason we do not have to encode definitions as part of the CNF. Abbreviations are assigned during an initialization phase, as described in the next Section (see also Alg. 2).

2.1 Initialization

After factoring out assumptions and adding abbreviations instead, every learned clause α contains *at most one* assumption or abbreviation. In this case we denote by $r(\alpha)$ this *replacement* literal. For other clauses we assume $r(\alpha)$ to be undefined. The graph represented by the definition map $\mathcal{G}$ can be interpreted as a (non-cyclic) circuit, which computes consistent values for abbreviations after all the assumption variables have been assigned. Special care has to be taken to handle assumptions and abbreviations, which are fixed by the user in between incremental calls. For instance, in MUS extraction, they are used to permanently select transition clauses [9] to be part of the extracted MUS.

```
Algorithm 1: assignAbbreviation
   Input: $\ell$: literal; var $\mathcal{I}$: interpretation; $\mathcal{G}$: definition map
1  $removeUnit(\mathcal{G}[\ell])$;
2  while $\mathcal{I}(\mathcal{G}[\ell])$ unassigned do
3      pick unassigned $\ell' \in \mathcal{G}[\ell]$;
4      assignAbbreviation($\mathcal{G}, \ell', \mathcal{I}$);
5  if $\mathcal{I}(\mathcal{G}[\ell]) = \bot$ then $\mathcal{I} \leftarrow \mathcal{I} \cup \{\neg\ell\}$ else $\mathcal{I} \leftarrow \mathcal{I} \cup \{\ell\}$;
```

In order to assign an abbreviation, we need to assign assumption variables and, recursively, every abbreviation in its definition. This is formulated in Alg. 1, which has the following arguments: the literal ℓ to be assigned, and (by reference) the current interpretation $\mathcal{I}$ and the definition map $\mathcal{G}$. First, literals assigned at the top-level (units), are removed from $\mathcal{G}[\ell]$. Next, while there is an unassigned literal ℓ' in $\mathcal{G}[\ell]$ and $\mathcal{G}[\ell]$ is itself unassigned by the current interpretation $\mathcal{I}$, we assign ℓ', using the same algorithm recursively. As soon as the value of $\mathcal{G}[\ell]$ under $\mathcal{I}$ is determined, we can also assign ℓ to $\mathcal{I}(\mathcal{G}[\ell])$.

By construction the definitions in the definition map $\mathcal{G}$ are non-cyclic. Further, we assume that every assumption is assigned by $\mathcal{I}$, as discussed in the previous section. Then this algorithm terminates and consistently assigns the value of each abbreviation ℓ to the value of its definition $\mathcal{G}[\ell]$.

2.2 Assigning the Set of Necessary Abbreviations

In the worst case, every learned clause resp. conflict requires a new abbreviation to be added. Therefore, in principle, the definition map grows linearly in the number of conflicts. This not only requires a huge amount of memory, but also needs substantial running time to initialize all the abbreviations of the definition map during incremental SAT calls.

However, since inactive [20] resp. less useful learned [21,22] clauses are frequently collected during the main CDCL loop of the SAT solver anyhow, many abbreviations turn out not to be referenced anymore after a certain point. They become *garbage abbreviations* and could be collected too. Actually, only the assignments of those abbreviations have to be initialized, which are still referenced in learned clauses (recursively). Assigning additional abbreviations is not harmful, but useless.

Algorithm 2 implements an initialization of abbreviations taking this argument into account. It returns an interpretation $\mathcal{I}$, which assigns all abbreviations recursively reachable from the clauses in the CNF Σ (which includes learned clauses). First, the algorithm initializes $\mathcal{I}$ by assigning all assumptions. Next, it traverses all clauses α to which a replacement $r(\alpha)$ has been added and then calls Alg. 1 to assign the replacement literal. The resulting $\mathcal{I}$ consistently assigns reachable abbreviations to the value of their definition in the definition map $\mathcal{G}$, unless a clause is found that has all its literals assigned to false.

Algorithm 2: initialization

```
Input: $\Sigma$: CNF formula; $\mathcal{A}$: assumptions; $\mathcal{G}$: a definition map
Result: $\mathcal{I}$ a partial interpretation
1 $\mathcal{I} \leftarrow \mathcal{A} \cup \{\text{top-level units}\}$;
2 foreach $\alpha \in \Sigma$ with $r(\alpha)$ defined do
3     if $r(\alpha)$ is unassigned by $\mathcal{I}$ then
4         assignAbbreviation($\mathcal{G}, r(\alpha), \mathcal{I}$);
5     if $\mathcal{I}(\alpha) = \bot$ then break;
6 return $\mathcal{I}$;
```

2.3 Assumption Core Analysis

As discussed in the introduction, applications of incremental SAT with assumptions often make use of the SAT solver's ability to return an *assumption core*, i.e., a subset of the given assumptions, which in combination with the given CNF can not to be satisfied. Intuitively, the assumption core exactly contains the assumptions "used" by the SAT solver to derive the inconsistency. In contrast to the concept of MUS, these assumption cores are typically not required to be minimal. As implemented in MINISAT [1] such an assumption core can be computed by a separate conflict analysis routine called "analyzeFinal", which recursively goes through the implication graph to only collect assumptions in contrast to the usual analysis routine of CDCL solvers which cuts off the search for a learned clause as soon as possible, e.g., following the 1st UIP scheme [23].

After factoring out assumptions and adding abbreviations the "analyzeFinal" procedure has to be adapted to care for abbreviations, which is described in Alg. 3. The algorithm takes as input a CNF formula Σ, the current unsatisfiable trail[2] $\mathcal{I}$, a clause α falsified under $\mathcal{I}$, the definition map $\mathcal{G}$, and returns the set of assumptions $\mathcal{C}$ "used" to establish the unsatisfiability proof. It starts by initializing $\mathcal{C}$ and the literals $\mathcal{V}$ already visited with the empty set. Next, the stack $\mathcal{T}$, containing the set of literals that must be further visited, is initialized with the conflict clause α. Then, while there is still an unvisited literal $\ell \in \mathcal{T}$, it is marked. Depending on its type three different cases have to be distinguished. First in line 5, if ℓ is an assumption, then ℓ is added to the conflict clause $\mathcal{C}$. Second in line 6, if ℓ is an abbreviation its definition $\mathcal{G}[\ell]$ is added to $\mathcal{T}$. This is actually the only part where the algorithm has to be adapted to recursively explore the definition map. Third in line 7, ℓ is neither an assumption nor an abbreviation and the reason of its propagation is added to $\mathcal{T}$ (implication graph exploration). Decision literals are assumed to have an empty set of antecedents.

[2] Every literal assigned to true, particularly those found during BCP, are added to a stack, called *trail*, to record the order of assignments. The reason, also called antecedent, of a forced assignment is saved too. Please refer to [1] for more details.

Algorithm 3: analyzeFinal

```
Input: $\Sigma$: CNF; $\mathcal{I}$: trail; $\alpha$: clause; $\mathcal{G}$: a definition map
Result: $\mathcal{C}$, a subset of the assumptions
1 $\mathcal{C} = \emptyset$; $\mathcal{V} = \emptyset$;
2 $\mathcal{T} \leftarrow \alpha$;
3 while $\exists \ell \in \mathcal{T} \setminus \mathcal{V}$ do
4     $\mathcal{V} \leftarrow \mathcal{V} \cup \{\ell\}$;
5     if $\ell$ is an assumption then $\mathcal{C} \leftarrow \mathcal{C} \cup \{\ell\}$;
6     else if $\ell$ is an abbreviation then $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{G}[\ell]$;
7     else $\mathcal{T} \leftarrow \mathcal{T} \cup reason(\ell, \mathcal{I})$;
8 return $\mathcal{C}$;
```

Example 1. Consider again the example in Fig. 2. Given $\{\overline{a_1}, \overline{a_2}, \overline{a_3}, \overline{a_4}, \overline{a_5}, \overline{a_6}\}$, learning α'_7 allows to conclude that the formula is unsatisfiable. Alg. 3 produces:

$\mathcal{T}$	$\mathcal{V}$	$\mathcal{C}$	ℓ
$\overline{p_2}, \ell_6$	$\emptyset$	$\emptyset$	$undef$
ℓ_6, ℓ_2	$\emptyset$	$\emptyset$	$\overline{p_2}$
ℓ_2, a_5, a_6	ℓ_5	$\emptyset$	ℓ_6
a_5, a_6, a_2, a_3	ℓ_5, ℓ_6	$\emptyset$	ℓ_2
a_6, a_2, a_3	ℓ_5, ℓ_6, a_5	a_5	a_5
a_2, a_3	ℓ_5, ℓ_6, a_5, a_6	a_5, a_6	a_6
a_3	$\ell_5, \ell_6, a_5, a_6, a_2$	a_5, a_6, a_2	a_2
$\emptyset$	$\ell_5, \ell_6, a_5, a_6, a_2, a_3$	a_5, a_6, a_2, a_3	a_3

The resulting learned clause is $(a_5 \vee a_6 \vee a_2 \vee a_3)$. Note, neither a_1 nor a_4 were actually "used" in deriving it. In the next section will make use of such an analysis to eagerly reduce the learned clause data base.

2.4 Reduce Learned Clause Database

Keeping all learned clauses slows down the SAT solver considerably. Thus heuristics to determine which learned clauses to keep resp. how and when to reduce the learned clause database are an essential part of state-of-the-art SAT solvers [20,21,22]. After an incremental SAT call returned "unsatisfiable", we propose to only keep those learned clauses, which were used to show that the assumed assumptions in this SAT call are inconsistent and discard all others.

Experiments in Sect. 3.2 will give empirical evidence for the effectiveness of these heuristics. Even though it is not a solid argument, an intuitive explanation could be that learned clauses are removed quite frequently anyhow. Further, most likely exactly those learned clauses related to the last set of assumptions are useful in the next SAT call too. This particularly applies to MUS extraction where the assumptions do not change much.

However, in order to apply these heuristics we need to be able to determine whether a certain clause was used in deriving the inconsistency. As it turns out, our definition map can be interpreted as partial proof trace for learned clauses (with assumptions) and thus gives us a cheap way to flush learned clauses and definitions not required to show that the given set of assumptions is inconsistent.

Algorithm 4: eagerLearnedClauseDatabaseReduction

```
  Input: var $\Delta$: set of learned clauses; var $\mathcal{G}$: a definition map; $\mathcal{V}$: literals;
1 foreach $\alpha \in \Delta$ do
2     if $r(\alpha)$ is an abbreviation and $r(\alpha) \notin \mathcal{V}$ then
3         $\Delta \leftarrow \Delta \setminus \alpha$;
4         remove $r(\alpha)$ and its definition from $\mathcal{G}$;
```

Focusing on the remaining relevant learned clauses and definitions in this "core" reduces run time, as our experiments in Sect. 3.2 will show.

Let us continue with Example 1 after learning α'_7. Only α'_2 and α'_7 are required to show unsatisfiability under the given set of assumptions, while α'_4 is not required and thus according to our heuristic should be removed. This eager reduction of the learned clause database can be easily implemented as a post-processing phase using $\mathcal{V}$ computed by analyzeFinal, which is shown in Alg. 4.

2.5 Assumption Aware Clause Minimization

New learned clauses can often be minimized by applying additional resolution steps with antecedent clauses in the implication graph. Two approaches are currently used to achieve this minimization: applying self-subsuming resolution, also called local minimization, or applying recursive minimization[24]. In recursive minimization several resolution steps are tried to determine whether a literal can be removed from the learned clause. In both cases resolutions are only applied if the resulting clause is a strict sub-clause. Sörensson and Biere [24] demonstrated that clause minimization usually improves SAT solver performance. In the following we will either apply this classical recursive minimization, no minimization at all, or a new form of recursive minimization, and thus do not consider local minimization further.

In the incremental setting with many assumptions, our preliminary experiments showed that classical clause minimization is not very effective. Usually the number of literals deleted in classical clause minimizations is rather small. As reason we identified the fact that assumptions are not obtained by unit propagation, and thus cannot be removed from learned clauses through additional resolution steps. Furthermore, non-assumption literals are often blocked by at least one assumption pulled in by resolution steps. The classical minimization algorithm requires that the resulting clause is a strict sub-clause. It is not allowed to contain more assumptions.

This situation is not optimal since assumptions, during one call of the incremental SAT algorithm, are assigned to false and can thus be considered to be irrelevant, at least for this call. Our new minimization procedure makes use of this observation and simply ignores additionally pulled in assumptions during minimization. The resulting "minimized" clause might even increase in size. However, it will never have more non-assumption literals than the original clause.

3 Experiments

The algorithms described above have been implemented within the SAT solver MINISAT [1], starting from the original version, used in the current version of the state-of-the-art MUS extractor MUSer [9]. It heavily makes use of incremental SAT solving with many assumptions following the selector variable-based approach [25]. Our modified version of [1] is called MINISAT$_{abb}$ (MINISAT with abbreviation). We focus on MUS extraction and compare the performance of MUSer for different versions of MINISAT.

For our experiments we used all 295 benchmarks from the MUS track of the SAT Competition 2011 [3] after removing 5 duplicates from the original 300 benchmarks. These benchmarks[4] have their origin in various industrial applications of SAT, including hardware bounded model checking, hardware and software verification, FPGA routing, equivalence checking, abstraction refinement, design debugging, functional decomposition, and bioinformatics. The experiments were performed on machines with Intel® Core$^{\text{TM}}$2 Quad Processor Q9550 with 2.83 GHz CPU frequency with 8 GB memory and running Ubuntu 12.04. Resource limits are the same as in the competition: time limit of 1800 seconds, memory limit of 7680 MB.

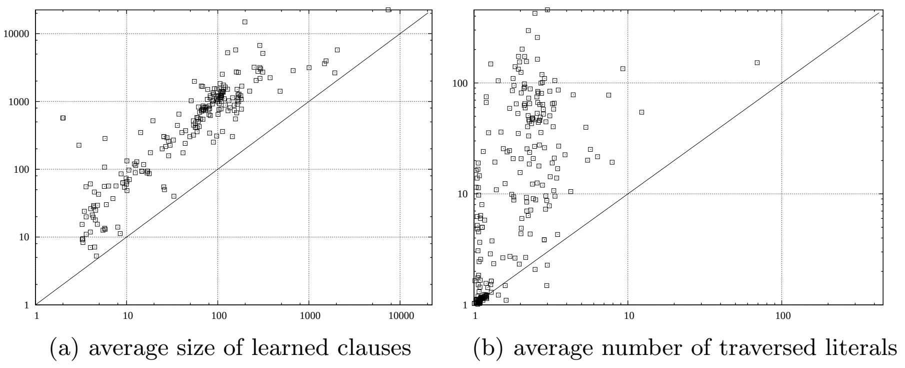
In the first experiment we apply our new approach of factoring out assumptions *without* changing clause learning. We then evaluate the impact of our new learned clause reduction scheme and our new clause minimization procedure. The experimental part concludes with more details on memory consumption.

3.1 Factoring Out Assumptions

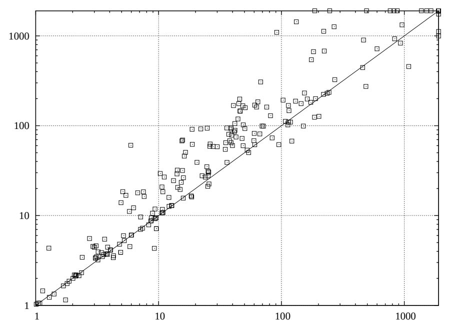
Fig. 3 shows a comparison between MUSer with our new approach based on factoring out assumptions, called MINISAT$_{abb}$, and the original version of MINISAT. First, in Fig. 3(a) the average size of learned clauses is compared. For many problems, adding clause abbreviations reduces the average size of learned clauses by an order of magnitude.

The main effect of our new technique is to reduce the size of learned clauses. This should also decrease the number of literals traversed while visiting learned clauses during BCP. In the scatter plot in Fig. 3(b) we focus on this metric and compare the average number of traversed literals while running both versions on the same instance. This includes the literals traversed in clauses visited during BCP, also including original clauses, but of course ignores clauses that are skipped due to satisfied blocking literals [26]. As the plot shows, the reduction in terms of the number of traversed literals is even more than the reduction of the average size of learned clauses. Consequently also the running time reduces considerably, see Fig. 3(c), but of course not in the same scale as in the previous plots. Note that in essence the "same clauses" are learned and thus the number of conflicts and learned clauses does not change.

[3] `http://www.satcompetition.org/2011`
[4] The set of benchmarks is available at `http://www.cril.univ-artois.fr/SAT11/`.



(a) average size of learned clauses (b) average number of traversed literals



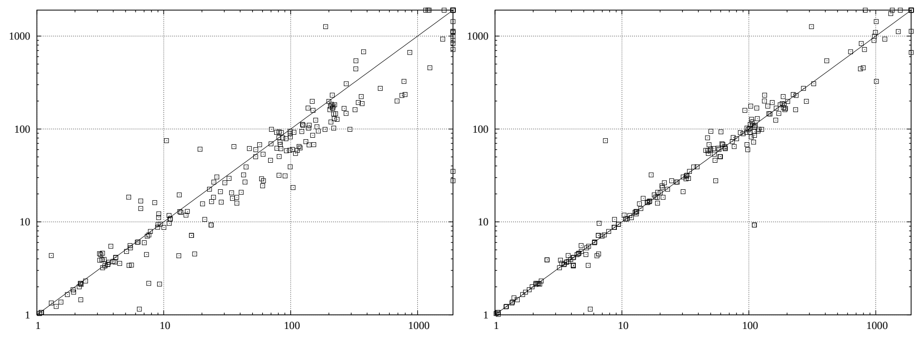
(c) running time (in seconds)

Fig. 3. Comparing MUSer on the 2011 competition instances from the MUS track, using the original MINISAT without abbreviations (y axis) vs. using our new version MINISAT$_{abb}$ with abbreviations (x axis) w.r.t. three different criteria.

The net effect of using abbreviations to factor out assumptions is that MUSer based on MINISAT$_{abb}$ solves 272 out of the 295 instances, and runs out of memory on 3 instances, whereas the version with the original MINISAT solves only 261 instances and runs out of memory in 13 cases. Our approach solves more instances, but not, at least primarily, because it runs out of memory less often.

As it turns out in the context of MUS extraction, definition clauses actually do not have to be watched. Further, abbreviation literals never have to be considered as decision and thus also do not have to be added to the priority queue (implemented as heap in MINISAT) for picking decisions. Thus we need initialization, by assigning all assumptions and abbreviations, the latter in incremental calls only, at the first decision level.

In order to make sure that the improvement observed in the previous experiment is independent from using our new optimized initialization phase, we report in Fig. 4(a) the run times of MUSer using the original version of MINISAT compared to the run times using a modified version of MINISAT, in which the assumption variables are assigned up-front and removed from the priority queue

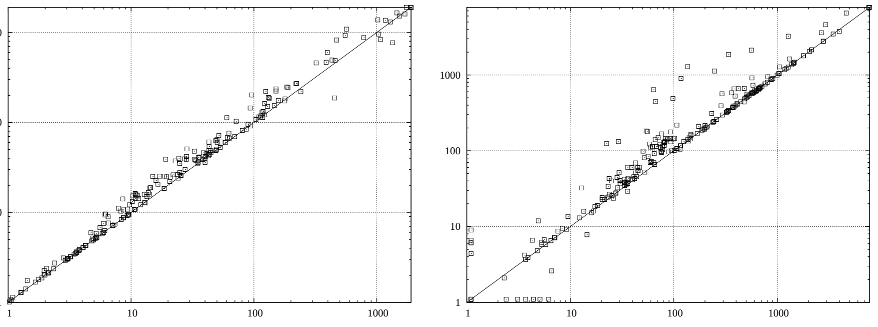


(a) MINISAT using initialization (b) MINISAT assumption-level-restarting

Fig. 4. On the left we show the running time of MUSer using MINISAT+init, a version of MINISAT, which initializes assumptions explicitly (x axis) vs. the original MINISAT version, which does not initialize them explicitly before search (y axis), both *without* abbreviations. Visiting each learned clause during initialization is time consuming without abbreviations. In the experiment shown on the right we only modified the restart mechanism to backtrack to the decision level of the last assigned assumption instead of backtracking to the root level. The modified version MINISAT+assumption-level-restarting (x axis) performs equally well as the original version of MINISAT (y axis). Running time is measured in seconds with a time limit of 1800 seconds as always.

initially too, called MINISAT+init. The results show that using this modified initialization scheme in the original version of MINISAT actually has a negative effect on the performance of MUSer (MUSer using MINISAT solves 261 instances whereas MUSer using MINISAT+init solves 257 instances) and thus can not be considered to be the main reason for the witnessed improvements in the first experiment. Our explanation for this effect is, that our initialization algorithm in essence needs only one pass over the learned clauses, even just a subset of all learned clauses, while initializing up-front BCP in MINISAT+init needs to visit lots of clauses during initialization. Note, again, that initialization has to be performed at the start of every incremental SAT call and might contribute a substantial part to the overall running time.

Modern SAT solvers based on the CDCL paradigm restart often by frequently backtracking to the root-level (also called top-level) [27,28,29,30] using a specific restart schedule [31,32,33,34]. With assumptions it seems however to be more natural to backtrack to the highest decision level, where the last assumption was assigned, which we call *assumption-level*. This technique is implemented in Lingeling [35], since it can naturally be combined with the technique of reusing the trail [36], but is not part of MINISAT. It might be conceivable, that forcing MINISAT to backtrack to the assumption-level during restarts can give the same improvement as initialization up-front. However, the experiment reported in Fig. 4(b) shows, that at least in MUS extraction, this "optimization" is useless.



(a) running time (in seconds) (b) memory used (in Mega Bytes)

Fig. 5. Comparison MUSer using MINISAT$_{abb}$ (y axis) vs. MINISAT$_{abb}$+g (x axis).

3.2 Learned Clauses Database Reduction

In this section, we study the impact on the performance of MINISAT$_{abb}$ w.r.t our new reduction algorithm for the learned clause database presented in Sect. 2.4. Fig. 5 compares MUSer using MINISAT$_{abb}$ with and without this more "eager garbage collector", which we denote by MINISAT$_{abb}$+g resp. MINISAT$_{abb}$. According to Fig. 5(b) eager garbage collection reduces memory consumption. Moreover, as shown in Fig. 5(a), this memory reduction does not hurt performance, since three more instances are solved (275 vs. 272) and only 1 instance (instead of 3) runs out of memory (see also Tab. 1).

3.3 Minimization of the Learned Clauses

In this section, we compare our new clause minimization procedure to existing variants of clause minimization. We consider three versions of MINISAT and MINISAT$_{abb}$ as back-end in MUSer [9]:

- without clause minimization (called *without*);
- the classical recursive clause minimization (*classic*) [24];
- our new clause minimization procedure (*full*) described in Sect. 2.5.

From the cactus plot in Fig. 6, which compares average size of learned clauses, we can draw the following conclusions. First, classical minimization is not effective in terms of reducing the average size of learned clauses, neither for MINISAT nor for MINISAT$_{abb}$, because it cannot remove assumptions during clause minimization. Classical minimization is slightly more effective with abbreviations than without. However, abbreviations might block self-subsumption during recursive resolution steps and thus prevent further minimization.

Next, we study the impact of our new full clause minimization described in Sect. 2.5 with and without using abbreviations. As reported in [24] for SAT solving *without* assumptions, recursive clause minimization typically is able to

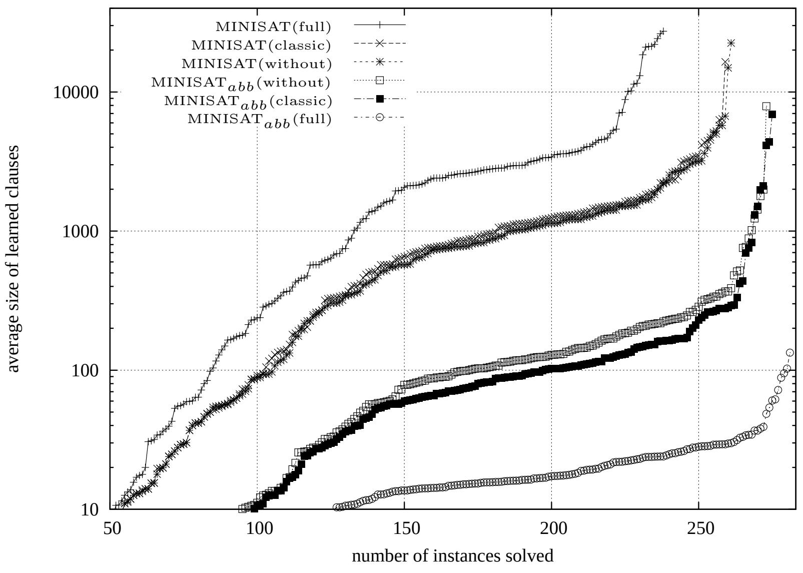


Fig. 6. Cactus plot reporting the average size of learned clauses produced by MUSer using MINISAT without abbreviations and MINISAT$_{abb}$ with abbreviations, and different clause minimization approaches. Factoring out assumptions (lower three curves) is always better than the original scheme (upper three curves). Further, full minimization gives a large improvement but only if assumptions are factored out. Without abbreviations full minimization actually turns out to be detrimental. Classic minimization only gives a small advantage over not using any minimization.

reduce the average size of learned clauses by one third. In the SAT solving *with* assumptions, as previously noted, assumptions prevent this reduction. With full clause minimization, however, we get back to the same reduction ratio of around 30% considering only literals that are neither assumptions nor abbreviations. Nevertheless, since deleting one literal is often necessary to apply additional resolutions, many new assumptions are added to the minimized clause. Using full minimization in MINISAT without abbreviations increases the average size of learned clauses by an order of magnitude, whereas MINISAT$_{abb}$ does not have this problem, since assumptions and abbreviations are factored out.

Actually, our new full clause minimization procedure in combination with MINISAT$_{abb}$ is able to reduce the average size of learned clauses by two orders of magnitude w.r.t the best version of MINISAT without abbreviations, while already one order of magnitude is obtained by MINISAT$_{abb}$ just by using abbreviations alone (with or without using classical clause minimization procedure).

In another experiment we measured the effect of our new garbage collection procedure Alg. 4. As it turns out, the average size is not influenced by adding this procedure, but as Tab. 1 shows, it has a positive impact on the number solved instances independent from the minimization algorithm used. Finally, this

	MINISAT	MINISAT$_{abb}$	MINISAT$_{abb}$+g
	#solved(MO)	#solved(MO)	#solved(MO)
without minimization	259(15)	272(3)	273(3)
classic minimization	261(13)	272(3)	275(1)
full minimization	238(25)	276(0)	**281(0)**

Table 1. The table shows the number of solved instances by MUSer within a time limit of 1800 seconds and a memory limit of 7680 MB, for different back-end SAT solver: the original MINISAT, then MINISAT$_{abb}$ with abbreviations, and finally MINISAT$_{abb}$+g with abbreviations and eager learned clause garbage collection. For each version of these three SAT solvers we further use three variants of learned clause minimization. The approach with abbreviations, eager garbage collection and full learned clause minimization, e.g. using all of our suggested techniques, works best and improves the state-of-the-art in MUS extraction from 261 solved instances to **281**.

tables also shows that the reduction of the average size of learned clauses directly translates into an increase of the number of solved instances. The combination of our new techniques improves the state-of-the-art of MUS extraction considerably.

3.4 Memory Usage

We conclude the experiments with a more detailed analysis of memory usage for the various considered versions of MUSer. As expected, Fig. 7 shows that shorter clauses need less memory. However, the effect in using our new techniques on overall memory usage is less pronounced than their effect w.r.t. to reducing average learned clause length. The main reason is that definitions have to be stored too. However, MINISAT with full clause minimization but without abbreviations produces a huge increase in memory consumption by an order of magnitude. This shows that factoring out assumptions is the key to make full clause minimization actually work. Also note, that our current implementation for storing definitions is not optimized for memory usage yet, and we believe that it is possible to further reduce memory consumption considerably.

4 Conclusion

In this paper we introduced the idea of factoring out assumptions, in the context of incremental SAT solving under assumptions. We developed techniques that work particularly well for large numbers of assumptions and many incremental SAT calls, as it is common, for instance, in MUS extraction. We implemented these techniques in the SAT solver MINISAT$_{abb}$ and showed that they lead to a substantial reduction in solving time if used in the SAT solver back-end of the state-of-the-art MUS extractor MUSer [9].

More specifically, experimental results show that factoring out assumptions by introducing abbreviations is particularly effective in reducing the average learned clause length, which in turn improves BCP speed. Even though memory

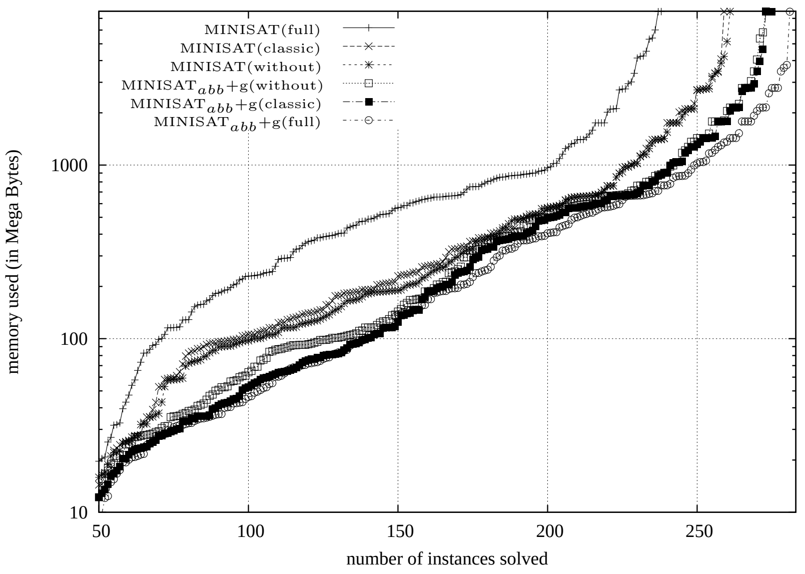


Fig. 7. Memory usage of MUSer based on the original MINISAT without abbreviations and MINISAT$_{abb}$+g with both abbreviations and eager garbage collection. Both versions of the MINISAT are combined with three different clause minimization strategies. Note, that even with eager garbage collection, which reduces memory consumption, e.g., see Fig. 5(b), the effect of our techniques on overall memory usage is not particularly impressive and leaves room for further optimization.

usage is not reduced at the same level as average learned clause lengths, using abbreviations leads to shorter running time. Furthermore, the ability to factor out assumptions is crucial for a new form of clause minimization, which gave another substantial improvement. In general, we improved the state-of-the-art in MUS extraction considerably.

Our prototype MINISAT$_{abb}$ uses rather basic data structures, which can be improved in several ways. Memory usage could be reduced by a more sophisticated implementation of managing abbreviations. Further, in the current implementation, identical definitions are not shared. A hashing scheme could cheaply detect this situation and would allow to reuse already existing definitions instead of introducing new ones. This should reduce memory usage further and also speed up the initialization phase.

Finally, it would be interesting to combine the techniques presented in this paper with more recent results on MUS preprocessing [10] and preprocessing under assumptions [19,37] resp. inprocessing [38]. We also want to apply our approach to high-level MUS extraction [7,8,16].

Software and more details about the experiments including log files are available at `http://fmv.jku.at/musaddlit`.

References

1. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. SAT'04. Volume 2919 of LNCS., Springer (2004) 502–518
2. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. ENTCS **89**(4) (2003) 543 – 560
3. Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model finding. In: Proc. CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications. (2003)
4. Marques-Silva, J., Lynce, I., Malik, S.: 4. In: Conflict-Driven Clause Learning SAT Solvers. Volume 185 of Frontiers in Artificial Intel. and Applications. IOS Press (February 2009) 131–153
5. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. **25**(2) (2012) 97–116
6. Grégoire, É., Mazure, B., Piette, C.: Extracting MUSes. In: Proc. ECAI'06. Volume 141 of Frontiers in Artificial Intel. and Applications., IOS Press (2006) 387–391
7. Nadel, A.: Boosting minimal unsatisfiable core extraction. In: Proc. FMCAD'10, IEEE (2010) 221–229
8. Ryvchin, V., Strichman, O.: Faster extraction of high-level minimal unsatisfiable cores. In: Proc. SAT'11. Volume 6695 of LNCS. (2011) 174–187
9. Belov, A., Marques-Silva, J.: Accelerating MUS extraction with recursive model rotation. In: Proc. FMCAD'11, FMCAD (2011) 37–40
10. Belov, A., Järvisalo, M., Marques-Silva, J.: Formula preprocessing in MUS extraction. In Piterman, N., Smolka, S., eds.: Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013). Volume 7795 of Lecture Notes in Computer Science., Springer (2013) 110–125
11. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Proc. SAT'06. Volume 4121 of LNCS. Springer (2006) 252–265
12. Andraus, Z.S., Liffiton, M.H., Sakallah, K.A.: Refinement strategies for verification methods based on datapath abstraction. In: Proc. ASP-DAC'06, IEEE (2006) 19–24
13. Marques-Silva, J., Planes, J.: On using unsatisfiability for solving maximum satisfiability. CoRR **abs/0712.1097** (2007)
14. Brummayer, R., Biere, A.: Effective bit-width and under-approximation. In: Proc. EUROCAST'09. Volume 5717 of LNCS. (2009) 304–311
15. Eén, N., Mishchenko, A., Amla, N.: A single-instance incremental SAT formulation of proof - and counterexample - based abstraction. In: Proc. FMCAD'10. (2010) 181–188
16. Nöhrer, A., Biere, A., Egyed, A.: Managing SAT inconsistencies with HUMUS. In: Proc. VaMoS'12, ACM (2012) 83–91
17. Huang, J.: Extended clause learning. AI **174**(15) (2010) 1277–1284
18. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning SAT solvers. In: Proc. AAAI'10. (2010)
19. Nadel, A., Ryvchin, V.: Efficient SAT solving under assumptions. In: Proc. SAT'12. (2012) 242–255
20. Goldberg, E.I., Novikov, Y.: BerkMin: A fast and robust Sat-solver. In: Proc. DATE'02, IEEE (2002) 142–149
21. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Proc. IJCAI'09, Morgan Kaufmann (2009) 399–404

22. Audemard, G., Lagniez, J.M., Mazure, B., Saïs, L.: On freezing and reactivating learnt clauses. In: Proc. SAT'11. Volume 7317 of LNCS., Springer (2011) 188–200
23. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: Proc. ICCAD'01. (2001) 279–285
24. Sörensson, N., Biere, A.: Minimizing learned clauses. In: Proc. SAT'09, Springer (2009) 237–243
25. Oh, Y., Mneimneh, M.N., Andraus, Z.S., Sakallah, K.A., Markov, I.L.: AMUSE: a minimally-unsatisfiable subformula extractor. In: Proc. DAC'04, ACM (2004) 518–523
26. Chu, G., Harwood, A., Stuckey, P.J.: Cache conscious data structures for boolean satisfiability solvers. JSAT **6**(1-3) (2009) 99–120
27. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In: Proc. AAAI/IAAI'98. (1998) 431–437
28. Huang, J.: The effect of restarts on the effectiveness of clause learning. In: Proc. IJ-CAI'07. (2007)
29. Pipatsrisawat, K., Darwiche, A.: RSat!2.0: SAT solver description. Technical Report Technical Report D153, Automated Reasoning Group, Comp. Scienc. Dept., UCLA (2007)
30. Biere, A.: Picosat essentials. JSAT **4**(2-4) (2008) 75–97
31. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters **47** (1993)
32. Biere, A.: Adaptive restart strategies for conflict driven SAT solvers. In: Proc. SAT'08. Volume 4996 of LNCS., Springer (2008) 28–33
33. Ryvchin, V., Strichman, O.: Local restarts. In: Proc. SAT'08. Volume 4996 of LNCS., Springer (2008) 271–276
34. Audemard, G., Simon, L.: Refining restarts strategies for SAT and UNSAT. In: Proc. CP'12. Volume 7514 of LNCS., Springer (2012) 118–126
35. Biere, A.: Lingeling and friends at the SAT Competition 2011. FMV Report Series Technical Report 11/1, Johannes Kepler University, Linz, Austria (2011)
36. van der Tak, P., Ramos, A., Heule, M.: Reusing the assignment trail in CDCL solvers. JSAT **7**(4) (2011) 133–138
37. Nadel, A., Ryvchin, V., Strichman, O.: Preprocessing in incremental SAT. In: Proc. SAT'12. (2012) 256–269
38. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Proc. IJCAR'12. (2012) 355–370