Journal on Satisfiability, Boolean Modeling and Computation 9 (2014) 27-51

Accelerated Deletion-based Extraction of Minimal Unsatisfiable Cores

Alexander Nadel alexander.nadel@intel.com
Design Technology Solutions Group,
Intel Corporation,
Haifa, Israel
Vadim Ryvchin vadim.ryvchin@intel.com
Design Technology Solutions Group, Intel Corporation, and
Information Systems Engineering, IE, Technion,
Haifa, Israel
Ofer Strichman ofers@ie.technion.ac.il
Information Systems Engineering,
IE, Technion,
Haifa, Israel

Abstract

Various technologies are based on the capability to find small unsatisfiable cores given an unsatisfiable CNF formula, i.e., a subset of the clauses that are unsatisfiable regardless of the rest of the formula. If that subset is irreducible, it is called a Minimal Unsatisfiable Core (MUC). In many cases, the MUC is required not in terms of clauses, rather in terms of a preknown user-given set of *high-level constraints*, where each such constraint is a conjunction of clauses. We call the problem of minimizing the participation of such constraints *high-level* minimal unsatisfiable core (HLMUC) extraction. All the current state-of-the-art tools for MUC- and HLMUC-extraction are *deletion-based*, which means that they iteratively try to delete clauses from the core. We propose nine optimizations to this general strategy, although not all apply to both MUC and HLMUC. For both cases we achieved over a 2X improvement in run time comparing to the state-of-the-art and a reduction in the core size, when applied to a benchmark set consisting of hundreds of industrial test cases. These techniques are implemented in our award-winning solvers HaifaMUC and HaifaHLMUC.

Keywords: *Minimal Unsat Core, Minimal High-Level Unsat Core, Resolution*

Submitted June 2014; revised November 2014; published November 2014

1. Introduction

Given an unsatisfiable CNF formula φ, an *Unsatisfiable Core* (UC) is any subset of φ that is unsatisfiable. The decision problem corresponding to finding the *minimum* UC is a Σ_2-

complete problem [22]. Finding a *minimal* UC — a UC such that the removal of any one of its clauses makes the formula satisfiable — is D^P-complete [37][1].

The problem for finding a *small*, a *minimal* (irreducible), the *minimum* (smallest minimal), or *all* the MUCs has been addressed frequently over the last decade [51, 19, 9, 36, 28, 49, 17, 13, 42, 8, 1, 27, 46, 18, 20, 38, 14, 26, 50, 33, 43, 12, 41, 6, 30, 7, 48, 5, 4, 29, 24, 35, 39] because of its theoretical and practical importance. The applications of MUC-extraction include abstraction refinement for model checking [31, 21, 4], formal equivalence verification [23, 12] and functional bi-composition [25, 10] — see [43, 34] for extensive surveys.

There are many uses to the core in SAT-based verification, typically related to abstraction or decomposition. In many cases, however, it is not the core C itself that is being used, rather C is processed further in order to check which *High-level Constraints* participate in the proof, where the grouping of clauses to high-level constraints is given as input to the problem. Hence, we can assume that in addition to the formula we are given as input a set of disjoint sets of clauses $HLC = \{H_1 \ldots H_m\}$ (High-Level Constraints), where each H_i is a set of clauses that together encode a high-level constraint. The goal is thus to find a core C that intersects a minimum number of constraints in HLC. This problem was first mentioned in [27], where an algorithm that finds *all* HLMUC-s was suggested, and later coined *the high-level minimal unsatisfiable core* problem[2] by the first author [34], who observed that in his experiments with industrial problems the number of clauses that belong to high-level constraints is on average about 5% of the clause database.

Two prominent examples of such techniques that are used in Intel and are described in more detail in the above reference are:

- A popular abstraction-refinement model-checking is based on iterating between a complete model checker and a SAT-based bounded model checker [31, 21]. The model checker takes an abstract model, in which some of the state variables are replaced with inputs, and either proves the property or returns the depth in which it found a counterexample. In the latter case, this depth is used in a bounded-model checking run over the concrete model, which may either terminate with a concrete counterexample, or with an unsat answer. In the latter case SAT's capability to identify an unsatisfiable core is used for identifying those state variables that are sufficient for proving that there is no counterexample at that depth. All the clauses that contain a given state variable (in any time-frame) constitute a constraint in HLC. Those state variables that participate in the proof define the next abstract model (these are the state variables that are *not* replaced by inputs), which is a refinement of the previous one. The process then reiterates until either the model checker is able to prove the property or the SAT solver finds a concrete counterexample.

- In formal equivalence verification (see, e.g., [23]), two similar circuits are verified to be functionally equivalent. This is done by decomposing the two circuits to 'slices' which are pair-wise verified for equivalence. The equivalence of each such pair is verified against various assumptions on the environment. In other words, rather than

[1] D^P is the class containing all languages that can be considered as the difference between two languages in NP, or equivalently, the intersection of a language in NP with a language in co-NP.

[2] This problem was called Group-Minimal Unsatisfiable Subsets in the 2011 SAT competition — the only competition so far in this category.

integrating a model of the environment with the equivalence verification condition, various properties of the environment are assumed, and added as constraints on the inputs of that condition. Then, if the equivalence is proven, it is still necessary to verify that the assumptions are indeed maintained by the environment. Each assumption is modeled with a set of clauses. The unsatisfiable core obtained when checking the equivalence is analyzed in order to find those assumptions that were used in the proof. Hence, here each constraint in HLC is a set of clauses that encode an environment assumption. Here too the verification process attempts to minimize the HLMUC in order to minimize the number of environment assumptions that should be verified.

The basic approach taken by all competitive MUC solvers and all HLMUC solvers as of [34] is *deletion-based*, which means that they iteratively try to delete clauses from the core until reaching minimality. This basic idea appeared in the context of linear programming in [11, 2], and adopted for CNF MUC-extraction in [33, 13]. In the initial *approximation stage* the algorithm finds a not-necessarily-minimal UC S with one or more invocations of a SAT solver [51, 19]. The second *minimization stage* applies the following deletion-based iterative process over S's clauses until S becomes a MUC. Each iteration removes a *candidate* clause c from S and invokes a SAT solver. If the resulting formula is satisfiable, c must belong to the MUC, so c is returned to S and marked as *necessary*. Otherwise c is removed from S. For the latter case, it was also proposed in [33, 13] to remove not only c but all the other clauses in S that were not required for the proof that c is not necessary. In addition, [33, 13] is using clause-sharing-based incremental SAT solving [45, 47] to speed-up the algorithm. Modern MUC extractors, HaifaMUC included, are based on this deletion-based algorithm, and are enhanced by model rotation [30, 6, 5], a technique that we describe later in Sect. 3.

One needs to keep track of the dependencies between clauses in the system in order to both extract the initial core at the approximation stage and be able to remove clauses at the minimization stage and return clauses to the system, whenever necessary. There exist two approaches to keep track of dependencies:

- **Resolution-based.** Many modern SAT solvers are capable of producing a resolution proof in case the formula is unsatisfiable. The approximation stage traverses the proof backwards from the empty clause, and reports the clauses at the leaves as the core [51, 19]. In the minimization stage, a candidate clause is temporarily removed together with its cone in the resolution proof, in order to check satisfiability without it.

- **Assumptions-based.** As of Minisat [16], many solvers support the *assumptions* technique. Assumptions are literals that are assigned TRUE as the first decisions. The approximation stage updates every clause with a new selector variable, whose negation is added to the list of assumptions. At the approximation stage the solver identifies which of those were required to prove unsatisfiability. The set of clauses corresponding to these assumptions constitute the initial UC. The minimization stage manipulates the values of the selector variables to temporary remove or return clauses.

It was shown in [1] that the resolution-based approach is faster than the assumption-based approach for finding one non-minimal UC, mainly because of the overhead of maintaining assumption literals in the assumption-based approach. The deletion-based algorithm for MUC-extraction can be implemented based on either the resolution- or the assumption-based infrastructure. Most of the improvements that we will present in this article can also be implemented in both, although we will only present them in the context of the former.

Contribution. This article merges and extends three earlier proceedings articles [34, 41, 35]. Based on [34], we introduce a deletion-based algorithm for finding a single MUC or HLMUC. The algorithm uses a single SAT instance for all invocations [15] and can be either resolution- or assumption-based. Based on [41, 35] we present nine improvements to the resolution-based MUC and HLMUC problems. Not all of these improvements are relevant and effective for both, as we will show. In contrast to [35], our presentation of the algorithm here is in the context of *incremental* SAT solving, which enables us to present all the algorithm in this article with a unified view. It gives a comprehensive picture of the techniques we use in our solvers HaifaMUC and HaifaHLMUC, both of which won the first place in the SAT'11 competition in the MUC/HLMUC tracks (no such competition was held since), and to the best of our knowledge are still the fastest available.

In the case of MUC we achieved with HaifaMUC a 55% reduction in run time comparing to MUSer2 [7] and solved 4% more instances, when running on the instances from the MUC track of the SAT'11 competition (MUSer2 is a deletion-based MUC solver, based on assumptions. We will describe it in more detail in future sections). In the case of HLMUC, we experimented with hundreds of industrial examples from Intel, and achieved a 55% reduction in run time comparing to a basic deletion-based algorithm, and a 28% improvement comparing to the assumptions-based technique described above. The configuration that achieves these improvements also reduces the core by 73% and 57%, respectively. In a different set of experiments, this time with the 197 GMUS competition benchmarks and with an additional optimization (rotation), we witnessed run-time which is 44% less than that of the latest version of MUSer2. More details on our experiments can be found in Sect. 4.1.

We begin in the next section by describing a basic deletion-based algorithm for MUC-extraction, and a variant for extracting HLMUC.

2. Resolution-based MUC and HLMUC

The improvements we consider are relevant to resolution-based core extraction. We implemented inside Minisat 2.2 a rather standard mechanism for maintaining the resolution DAG. The resolution information is kept in a separate database, which we will call here the *resolution table*. This table maintains the indices of the parents and children of each derived clause. On top of this we implemented the reference counter technique of Shacham et al. [42]. In this technique every conflict clause has a counter, which is increased every time it resolves a new clause, and decreased when a child clause is erased. Once the counter of a clause is 0, it does not need to be maintained any longer for the purpose of later retrieving the resolution DAG. It is removed from the resolution and the counter of the parent is reduced by 1, which may create a chain of reductions in the resolution graph. In the experiments that were reported in [42] this optimization led to a reduction by a factor of 3 to 6 in the size of the resolution table.

The unsatisfiable core is retrieved as usual by backward traversal from the empty clause to the roots. But since we are interested in minimizing the core, the story does not end here. Consider the MUC algorithm that appears in Alg. 1. It maintains a set M, initialized to $\emptyset$, which in the end of the algorithm holds a MUC of the input formula. The algorithm simply iterates once over the set of clauses and checks which can be removed without making the formula satisfiable. Each time it succeeds in removing a clause c, it resets its starting point to the new proof in order to accelerate termination. This optimization was introduced in [33, 13] and called *clause-set refinement* later in [5].

All the algorithms that we present in this article are geared towards incremental solving, which is the reason we give the SAT solver a proof π, in the form of a resolution graph ending with the empty clause, rather than the original formula Ψ. $cone(c, \pi)$ denotes the cone of a clause c in π (i.e., the part of Π that is reachable from c. By definition it must hold c itself and the empty clause $\perp$), and $core(\pi)$ denotes the unsatisfiable core of π, namely all input clauses that belong to π. We assume that the SAT call SAT returns a tuple $\langle IsSAT, \pi\rangle$, where $IsSAT$ is the result, and π, in case the result is false (UNSAT), is a proof of unsatisfiability. We assume that this proof is 'trimmed', meaning that it only includes nodes that can reach the empty clause and the edges between them. Other clauses that were learned in the solution process are removed.

Algorithm 1 Resolution-based MUC-extraction with clause-set refinement.

```
Input:   Unsatisfiability proof $\pi$ of $\Psi$.
Output:  A MUC of $\Psi$.

$IsSAT := false$;
$M := \emptyset$;
while ($true$) do
    if ($IsSAT$) then
        $M := M \cup \{c\}$;
    if $core(\pi) = M$ then break;
    Choose $c \in (core(\pi) \setminus M)$;
    $\langle IsSAT, \pi\rangle := SAT(\pi \setminus cone(c, \pi))$        ▷ $cone(c, \pi)$ is the cone of $c$ in $\pi$
return $M$;
```

Let us now shift our focus to the HLMUC problem. We present a basic deletion-based algorithm for this problem in Alg. 2. The input to this algorithm is a proof π of a formula Ψ of the form:

$$\Psi = \bigwedge_{H_j \in HLC} H_j \wedge \Omega$$

where $HLC = \{H_1 \dots H_m\}$ is a set of high-level constraints, each of which is a set (or a conjunction, depending on the context) of clauses, and Ω is a standard CNF formula called the *remainder*. The set HLC itself is also an input to this algorithm. The output of the algorithm is a subset $HLC' \subseteq HLC$ such that $\Psi' = \bigwedge_{H_j \in HLC'} H_j \wedge \Omega$ is unsatisfiable, and no constraint can be removed of HLC' without making Ψ' satisfiable.

Algorithm 2 Resolution-based HLMUC-extraction.

Input: Unsatisfiability proof π of $\Psi = \bigwedge_{H_j \in HLC} H_j \wedge \Omega$, and a set HLC.
Output: A HLMUC with respect to HLC and Ω.

```
1:  $IsSAT := false$;
2:  $M := \emptyset$;
3:  while (true) do
4:      if ($IsSAT$) then
5:          $M := M \cup \{H_k\}$;
6:      else
7:          for ($H_i \in HLC$) do
8:              if ($H_i \cap core(\pi) = \emptyset$) then
9:                  $\pi := \pi \setminus cone(H_i, \pi)$;
10:                 $HLC := HLC \setminus \{H_i\}$;
11:     if $HLC = M$ then break;
12:     Choose $H_k \in (HLC \setminus M)$;
13:     $\langle IsSAT, \pi \rangle := SAT(\pi \setminus cone(H_k, \pi))$;    ▷ If unsat, $\pi$ is assigned the new proof
14: return $M$;
```

The algorithm checks the necessity of each constraint in HLC for the proof, either once or not at all. If it finds it necessary (i.e., without it the formula becomes satisfiable), it adds it to a set M, which is initialized to $\emptyset$. Consider first the case that the formula is satisfiable (note that this is never the case in the first iteration): in such a case the constraint H_k that was chosen to be checked (lines 12– 13) is simply added to M in line 5. Now consider the case that the formula is unsatisfiable, and a constraint H_i that does not participate in the proof π. The algorithm removes H_i and its cone from π, and H_i from the set HLC. Note that H_i will never be checked again. Also note that the condition in line 8 is guaranteed to be satisfied for the recent H_k chosen in line 12, because it cannot be part of the core M (see line 13). This implies that in the case of UNSAT at least one element is removed from HLC.

The termination argument is subtle. In each iteration of the main loop, the algorithm either removes elements of HLC (lines 7–10), or adds an element H_k to M (line 12). In the latter case H_k is guaranteed to stay in HLC until the end of the algorithm, because by definition H_k is in the core of every proof and will therefore never satisfy the condition in line 8. Together with the fact that M is initialized to $\emptyset$ (line 2), this guarantees that the two sets HLC and M are eventually equal, which guarantees termination.

It is interesting to note that Alg. 2 is tailored for HLMUC and not for MUC. The difference is evident by observing that if a constraint H_i participates in the proof then its entire set of clauses is retained in subsequent attempts to remove other constraints. For example, if $H_i = \{c_1, c_2\}$, and only c_1 participates in the proof, Alg. 2 retains both c_1 and c_2, because removing c_2 does not reduce the size of the HLMUC, whereas it may assist in consecutive iterations. Furthermore, retaining c_2 is necessary in order to guarantee minimality. Without it we may miss the fact that some other constraint can be removed.

Table 1. The nine optimizations covered in Sect. 3 and their relevance to the two goals MUC and HLMUC. Optimizations G – I are applicable to only one of the goals (at least in their basic form), as we explain in the text that describes these optimizations.

	Optimization	MUC	HLMUC	MUC-Biased
A.	Maintaining partial resolution proofs	+	+	
B.	Selective clause minimization	+	+	✓
C.	Postponed propagation over HLC-clauses	+	+	✓
D.	Reclassifying HLC-clauses	+	+	✓
E.	Selective learning of HLC-clauses	+	+	✓
F.	Selective Chronological backtracking	+	+	✓
G.	A removal strategy		+	
H.	Eager model rotation	+	+	
I.	Path strengthening	+		

It is not hard to see that Alg. 1 is a special case of Alg. 2, in which every clause is a high-level constraint. Indeed, in such a case HLC is equivalent to $core(\pi)$, and the condition in line 8 of Alg. 2 is equivalent to a clause not being reachable from $\bot$. Since we aim at a uniform presentation of the algorithms for both problems, this observation is important. We will use the notation HLC to refer to the high-level constraints in case of solving the HLMUC problem, but the reader should keep in mind that the same algorithm can be used for MUC-extraction as is, by referring to each clause in the core as a separate set in HLC.

3. Optimizations

In this section we describe nine optimizations to the basic algorithm that was presented in the previous section. Their relevance to our two goals, MUC and HLMUC, is summarized in Table 1.[3.] Most of these optimizations — see the rightmost column in the table — bias the search towards proofs that use a smaller core or a high-level core. The other optimizations only shorten run-time. Optimizations A − G were first introduced in [41], whereas optimizations H,I first appeared in [35].

We will use the following terminology: a clause is an *HLC-clause* if it either belongs to one of the initial constraints in HLC or is a descendant of such a clause in the resolution DAG. Other clauses are called *remainder* clauses. We say that a literal is *HLC-implied* if it is implied by an HLC-clause, and just *implied* otherwise.

A: Maintaining partial resolution proofs. In this optimization we maintain only clauses in the cone of HLC-clauses in the resolution table, and the links between them. That is, we save an HLC-clause, and the parents and children that are also HLC-clauses. Comparing to full resolution, this reduces the amount of memory required by more than an order of magnitude in most cases, reduces the amount of time that it takes to find

[3.] Optimizations E – F apply to both MUC and HLMUC, but experiments show that they have negligible effect with MUC, which is the reason that they were not reported in our earlier proceedings version [35].

clauses that are in the cone of an HLC (recall that in line 9 of Alg. 2 HLC-clauses are removed together with their cones), and, more importantly, allows to activate a certain simplification (see next paragraph) for remainder clauses, which otherwise has to be turned off when running Alg. 2.

The simplification we are referring to is applied at decision level 0. If the clause database includes a unit clause, e.g., (x), then many solvers would remove those clauses that contain x, and remove $\neg x$ from all other clauses, at decision level 0 (MiniSat is a little different in this respect: it does not remove $\neg x$ from existing clauses once x is learned, but rather it does not add $\neg x$ to new learned clauses). This simple, yet powerful simplification has to be turned off in an incremental setting, as in Alg. 2, or else the connection between the unit clause and the clauses it subsumed or reduced has to be maintained. The reason is that (x) may be later on removed, and hence the simplifications have to be undone. Since in practice this extra book-keeping is not cost-effective, such simplification is typically turned off in incremental setting. For remainder clauses, however, we can use this simplification, since we know that these clauses are not going to be removed in future instances, and hence no extra information needs to be saved.

B: Selective clause minimization. Clause minimization [3, 44] is a technique for shrinking conflict clauses. Once a clause is learnt, each of its literals is tested: if it implies other literals in the clause, it can be removed.

Example 1 *Consider the following clauses:*

$$\begin{array}{lllllllll} C_1 & = & (\neg v_1 \vee v_2) & C_2 & = & (\neg v_2 \vee v_3) & C_3 & = & (\neg v_4 \vee v_5) \\ C_4 & = & (\neg v_5 \vee v_6) & C_5 & = & (\neg v_1 \vee \neg v_3 \vee \neg v_4 \vee \neg v_6) & & & \end{array}$$

Suppose that the first decision is v_1. This decision implies v_2 (from C_1) and v_3 (from C_2). Suppose now that the next decision is v_4. This decision implies v_5 (from C_3) and v_6 (from C_4) and a conflict in clause C_5. Conflict analysis based on 1-UIP returns in this case a new clause $C = (\neg v_1 \vee \neg v_3 \vee \neg v_4)$. From C_1 and C_2 we can see that $v_1 \rightarrow v_3$, or equivalently $\neg v_3 \rightarrow \neg v_1$, which is an implication between literals in C. Clause minimization will find this implication by following the resolution DAG and remove $\neg v_3$. ∎

We will not present the full algorithm for clause minimization here, but rather only mention that it is based on traversing the resolution DAG backward from each literal l in the learned clause. The hope is to hit a 'frontier' of other literals from the same clause that by themselves imply l. If in this process we hit a decision variable, it means that l cannot be removed.

Example 2 *Continuing the previous example, the algorithm scans each non-decision literal in C. Consider v_3: this literal was implied in C_2, and hence we progress to look at the other literal in that clause, namely v_2. This literal was implied by C_1 and hence we look at v_1. But since $v_1 \in C$, it means that we found an implication within C, and hence $\neg v_3$ can be removed. Note that the minimized clause can be resolved from the original one and the clauses that are traversed in the process. In this case*

$$Res(C, Res(C_1, C_2)) = (\neg v_1 \vee \neg v_4) \; .$$

∎

The problem with clause minimization in our context is that it may turn a non-HLC-clause C into a shorter HLC-clause C'. This can happen if the minimization process uses an HLC-clause: in that case C' has to be marked as an HLC-clause as well. Furthermore, it can turn an HLC-clause C that depends on a certain set of high-level constraints, into a shorter HLC-clause that depends on *more* such constraints. This means that if that clause will participate in the proof, it will 'pull-in' more constraints into the core.

Our suggested optimization is to cancel clause minimization in any case that an HLC-clause is involved. In other words, we prefer a large clause that depends on a few constraints, over a smaller one with more such dependencies. The latter may pull more constraints into the proof, and lead to other such clauses. We aspire, instead, to keep the resolution table as small as possible and with the fewest connections to HLC-constraints. Ideally we should check whether using a certain HLC-clause in the minimization process indeed adds dependencies, but this is simply too expensive: for this we would need to traverse the DAG backwards all the way to the roots in order to check which constraints are involved.

It is interesting to analyze the behavior of the assumptions-based method with respect to clause minimization. It turns out that it solves this problem for free, and hence in this respect it is a superior method. In fact from analyzing various cases in which it performs much better than the clause-based method (before the optimizations suggested here were added), we realized that this is the main cause for the difference in run-time, rather than the facts mentioned in the introduction (the fact that it does not need to save the resolution table, nor to extract the core in the end of each iteration). How does it solve this problem for free? Observe that with this technique all HLC-clauses have as literals all the selector variables that correspond to constraints that were used in deriving that clause. For example, let H_1, H_2 be two constraints with associated selector variables l_1, l_2 respectively. If H_1 and H_2 participate in inferring C, then C must contain $\neg l_1$ and $\neg l_2$. This is implied by the fact that selector variables appear only in one phase in the formula, and hence cannot be resolved away. Hence the presence of these literals in HLC-clauses is an invariant. If we falsely assume that a minimized clause C can increase its dependency on constraints, we immediately reach a contradiction: the supposedly added constraint implies that a new selector variable was added to C, which contradicts the fact that literals are only removed from C in the minimization process.

C: Postponed propagation over HLC-clauses. In this optimization we control the BCP order. We first run BCP over non-HLC-clauses until completion. If there is no conflict, we propagate a single implication due to an HLC-clause, and run regular BCP again. We repeat this process until no more propagations are possible or reaching a conflict. The idea behind this optimization is to increase the chances of learning a remainder clause rather than an HLC-clause.

The way we implement it is the following: during BCP, every time we discover a new implication through an HLC-clause (i.e., that clause is the antecedent), instead of adding it to the assignment stack, we add it to a separate queue called *HLCImplicationsQ*. When BCP over the assignment stack ends (without a conflict), then we copy the first element of *HLCImplicationsQ* to the assignment stack and reactivate BCP. We continue this process until either reaching a conflict or *HLCImplicationsQ* is empty.

D: Reclassifying HLC-clauses. The SAT call in line 13 of Alg. 2 involves removing temporarily the cone of an HLC-constraint H. When the result is SAT, we add its clauses back as *remainder* clauses, together with all the clauses in its cone that do not depend on other constraints. To identify this set of constraints, we employ an algorithm in the style of a least-fix-point computation. We insert all the H clauses into a set S. Then we add all the children of those clauses that all their parents are in S. We repeat this process until reaching a fix-point.

Without this optimization H's clauses are added back as is, with their marking as HLC-clauses. By adding them back as remainder clauses, we enable more simplifications, such as propagation of unit clauses at decision level 0 (we described this simplification as part of optimization A). In fact if a clause is indeed not in any cone of a constraint in HLC, then it benefits most of the optimization that we describe here to have it marked as a remainder clause.

E: Selective learning of HLC-clauses. When detecting a conflict, the learned clause may be an HLC-clause. If all else is equal, such a clause is less preferable than a remainder clause, as it may increase the HLMUC, in addition to the fact that it leads to a larger resolution table and hence longer run times. We found that learning a non-asserting remainder clause instead, combined with partial restart, improves the overall performance. The learning of the remainder clause is essential for termination, and also turns out to decrease run time. The alternative remainder clause that we learn is even closer to the conflict than the first UIP. We can learn it only if the conflicting clause is not an HLC-clause; in other cases we simply revert to learning the HLC-clause. Learning the remainder clause is done by reanalyzing the conflict graph *as if the HLC-implications were decisions*. This optimization is only ran in conjunction with optimizations B and C above, for reasons that we will soon clarify. Alg. 3 describes the procedure for learning this clause.

Algorithm 3 An algorithm that attempts to find a remainder conflict clause by reanalyzing the conflict graph as if the HLC-implications were decisions. Returns a remainder clause if one can be found, and NULL otherwise.

```
function Get_Remainder_Clause
   1. If the conflicting clause is an $HLC$-clause then return NULL.
   2. Search an $HLC$-implied literal $l$ in the trail, starting from the latest implied literal
      and ending just before the 1-UIP literal.
   3. Convert the implication of $l$ into a decision, and update accordingly the decision level
      of all implied literals in the trail that come after it.
   4. Call Analyze_Conflict() with the same conflicting clause, but while referring to
      the new decision levels. Let $C$ be the resulting conflict clause.
   5. Return $C$.
```

Note that the fact that we use this algorithm only when optimization C is active, guarantees that the literals searched and updated in steps 2 and 3 are implied by l, i.e., the fact that BCP was ran to completion on non-HLC-clauses before asserting l, guarantees that the rest of the implications at that decision level depend on asserting l. Also note that the clause learnt in step 4 is necessarily a remainder clause because Analyze_Conflict() cannot cross an HLC-implied literal (such implications were made into decisions), and that it corresponds to a cut in the implication graph to the right of the first UIP. The reason we activate this optimization in conjunction with optimization B, is that we want to refrain from a case in which we learn a remainder clause, but it then turns into an HLC-clause owing to clause minimization. This is not essential for correctness, however: we could also have just compared this smaller HLC-clause to the original one and choose between the two, but our experience is that it is better to give priority to minimizing the number of HLC-clauses. Finally, note that there is no reason to revert the changes made to the trail, because backtracking removes this part of the trail anyway.

Example 3 *Figure 1 presents an implication graph, where HLC-implications are marked with dashed edges. The marked 1-UIP cut in the top drawing is calculated while considering such implications as any other implication. The suggested heuristic is to learn instead a normal clause, by considering such implications as new decisions, as depicted in the bottom drawing.* ▟

As mentioned earlier, learning the alternative clause is combined with a partial restart. Let dl be the level to which we would have jumped had we learned the HLC-clause. We backtrack to dl, but at this point nothing is asserted because we did not learn an asserting clause. We then move to the next decision level, $dl{+}1$, and decide the negation of the original 1-UIP literal. Hence instead of learning an asserting clause and implying the negation of the 1-UIP literal, we refrain from learning that clause and *decide* on the same value.

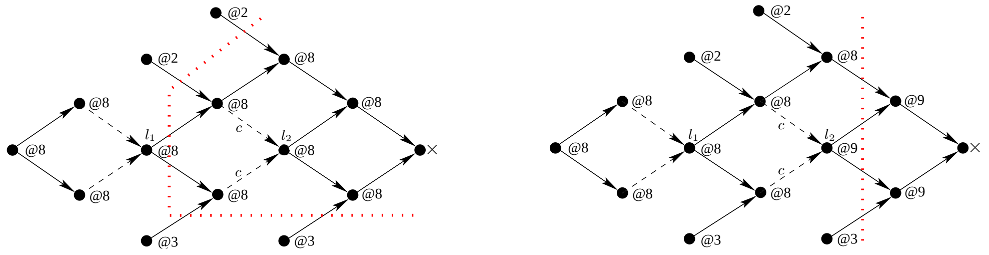
This assignment is neither necessary nor sufficient for preventing the same conflict to occur. What prevents us from entering an infinite loop in the absence of standard learning is the fact that we learn at least one clause between such partial restarts. In the presence of clause deletion, however, this argument generally does not hold, so we cannot guarantee termination in all cases, although we never witnessed entering a loop in practice[4.].

We can still argue for termination, however, even in the presence of clause deletion[5.]: Our procedure is almost equivalent to the normal sequence of learning an asserting clause, backtracking to the asserting level dl, performing propagation, and (possibly) erasing the clause. The only difference is that rather than asserting the literal at dl, we decide its value at level $dl + 1$. The termination argument in a normal solver is that we cannot enter a decision level twice with the same partial assignment (since we backtracked and assigned at least one variable that was not assigned earlier); here we can make the same argument about decision level $dl + 1$, because at that level we decide the 1-UIP literal, whereas its negation was *implied* earlier.

Example 4 *Referring again to the conflict graphs in Example 3, our solver backtracks to the end of level 3 — the same level we would have jumped with the original HLC-clause — progress to level 4 and decides $\neg l_1$.* ▟

[4.] Nontermination is not uncommon in modern solvers. Several solvers combine restarts with a non-increasing gap and clause deletion, which can lead to nontermination. Entering a loop is extremely rare in practice, however.

[5.] For the sake of discussion assume that there are no restarts. Arbitrary restarts combined with clause deletion can make any solver non-terminating.



(a) (b)

Figure 1. In these conflict graphs, dashed arrows denote HLC-implications, and the dotted lines denote 1-UIP cuts. In (a), where such implications are referred to as any other implications, the learned 1-UIP clause must be marked as an HLC-clause, since it is resolved from the HLC-clause c. We can learn instead a normal clause by taking, for example, the 1-UIP clause in the conflict graph (b). In that graph, c's implications are considered as decisions, which changes the decision levels labeling the nodes.

In our experiments we also tried other decisions (such as $\neg l_2$ in the example above), but $\neg l_1$ seems to work better in practice. We also tried different strategies of updating the scores. The best strategy we found in our experiments is to update the score according to both the original and the alternative clause.

F: Selective Chronological backtracking. Recall that optimization E involves a partial restart when learning an HLC-clause. Different heuristics can be applied in order to choose the backtracking level. Our experiments show that if we only backtrack one level, rather than to the original backtrack level as explained above, the results improve significantly. The complete set of data, available from [40], shows that in most instances this heuristic improves the run time; moreover, it reduces the number of conflicts, which implies that it improves the search. It seems that the reason for the success of this heuristic is related to the fact that with the normal backtracking and score scheme we may lose the connection to the clause that we actually learn, i.e., the scores might divert the search from a space which is more relevant to the alternative clause that we learn.

G: A removal strategy. Recall that in line 12 of Alg. 2 constraints are removed in an arbitrary order. We suggest a simple greedy heuristic instead for HLMUC: remove the high-level constraint that contributed the largest number of clauses to the proof. This heuristic, as will be evident in the next section, reduces the size of the resulting core but slightly increases run time.

We also experimented with a heuristic by which we remove the constraint with the *least* number of clauses in the proof, speculating that this leaves more clauses in the formula and hence increases the chance that there will be a proof without this constraint. This option also improves performance comparing to the arbitrary order with which we started, but is not as good as the one suggested above. There is an indirect cause behind this difference: the large constraints (i.e., those that have many clauses) are typically necessary for the proof regardless of the other constraints, and hence the faster we make them remainder constraints — with optimization D — the faster the rest of the solution process is. This, in turn, affects the size of the core because it leads to less time-outs. As we will explain in the next section, the result of the algorithm when interrupted by a time-out is the last computed core, or, in case that even the first iteration does not terminate, the entire set of HLC-clauses.

H: Eager Model Rotation Model rotation [30, 6, 5] can improve deletion-based MUC-extraction by searching for additional clauses that should be marked as necessary *without* an additional SAT call. Suppose, for example, that for an unsatisfiable set S, $S \setminus c$ is satisfiable. Consequently c is marked as necessary. Let h be the satisfying assignment. Note that $h(c) = \text{FALSE}$, because otherwise $h(S)$ would be TRUE, which contradicts S's unsatisfiability. Now, suppose that an assignment h' that is different than h in only one literal $l \in c$ satisfies all the clauses in S other than exactly one clause $c' \in S$. Hence $h'(S \setminus c') = \text{TRUE}$, which means that like c, c' must also be in any unsatisfiable subset of S, and can therefore be marked as necessary as well. Rotation flips the values of each of c's literals one at a time in search of such clauses. When one is found, rotation is called recursively with c'. This algorithm is summarized in Fig. 2(a). We observe that rotation, originally proposed in the context of assumption-based MUC-extraction, can be integrated into our resolution-based algorithm without any changes.

Fig 2(b) shows ERMR (Eager Recursive Model Rotation) — an improvement to rotation that weakens rotation's terminating condition. The reader may benefit from first reading the main algorithm in Alg. 4, which calls ERMR. The only difference between ERMR and RMR is that ERMR may call rotation with a clause that is already in M, the reason being that it can lead to additional marked clauses owing to the fact that the call is with a different assignment. Clearly there is a tradeoff between the time saved by detecting more clauses for M and the time dedicated to the search. For example, one may run RMR with more than one satisfying assignment as a starting point, but this will require additional SAT calls to find extra satisfying assignments. ERMR refrains from additional SAT calls. Rather it changes the stopping criterion: instead of stopping when $c \in M$ (line 4 in Fig. 2(a)), it stops when $c \in K$, where K holds the clauses that were discovered in the *current* call from MUC. There are other variations on weakening the terminating condition of rotation in the literature [5, 48]. We leave to future study a detailed comparison of our algorithm to these works.

Rotation for HLMUC. Rotation and eager rotation are irrelevant in their basic form to HLMUC, since flipping one literal does not guarantee the satisfaction of the entire set of clauses in the removed high-level constraint $H \in HLC$. We therefore apply the following strategy: we find the set of literals in the intersection of all the clauses in H that are unsatisfied by the current assignment. Flipping the assignment of each of these literals satisfies H by construction. We then check if it happens to contradict a single high-level constraint $H' \in (HLC \setminus H)$. If yes, then H' is necessary and therefore added to M. For comparison MUSer2 also applies rotation (not eager rotation), but only when a single clause in H is unsatisfied.

I: Path Strengthening. Path strengthening relies on the following property, which we call *cut falsifiability* (observed already in [13, 33]). Let S be an unsatisfiable formula, π its resolution proof, and c a candidate clause. Then, any model h to $S \setminus cone(c, \pi)$ must falsify

Algorithm 4 Deletion-based MUC-extraction enhanced by eager rotation.

Input: Unsatisfiability proof π of Ψ.
Output: A MUC of Ψ.

```
$IsSAT := false$;
$M := \emptyset$;
while ($true$) do
    if ($IsSAT$) then
        $K := \{c\}$;
        $M := ERMR(\Psi, c, M, K, h)$                ▷ $h$ is the satisfying assignment
    if $core(\pi) = M$ then break;
    Choose $c \in (core(\pi) \setminus M)$;
    $\langle IsSAT, \pi \rangle := SAT(\pi \setminus cone(c, \pi))$     ▷ $cone(c, \pi)$ is the cone of $c$ in $\pi$
return $M$;
```

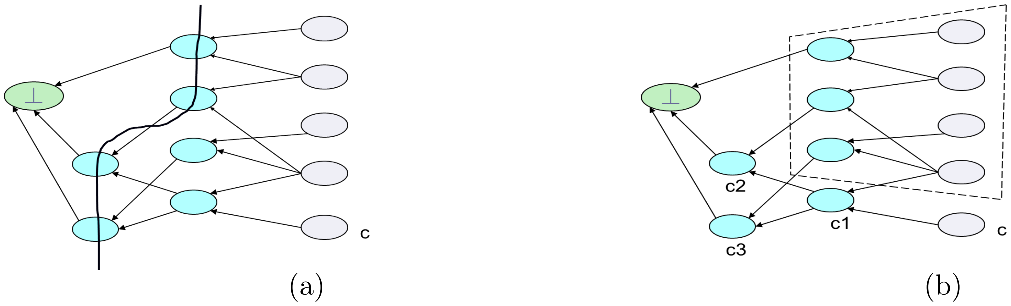
```
function RMR($S, M, c, h$)
  for all $x \in Var(S)$ do
    $h' := h[x \leftarrow \neg x]$; ▷ swap assignment
    if $UnsatSet(S, h') \equiv \{c'\} \wedge c' \notin M$ then
      $M := M \cup \{c'\}$;
      RMR($S, M, c', h'$);
```

(a)

```
function ERMR($S, M, K, c, h$)
  for all $x \in Var(S)$ do
    $h' := h[x \leftarrow \neg x]$;
    if $UnsatSet(S, h') \equiv \{c'\} \wedge c' \notin K$ then
      $K := K \cup \{c'\}$;
      if $c' \notin M$ then $M := M \cup \{c'\}$;
      ERMR($S, M, K, c', h'$);
```

(b)

Figure 2. (a) The recursive model rotation of [6]. Input: S is an unsatisfiable set, M is the current core, c is the most recent candidate clause (whose removal makes the formula SAT), and h is an assignment. $UnsatSet(S, h')$ is the subset of S's clauses that are unsatisfied by the assignment h' (b) our modified version. K is a set of clauses that is initialized to c before calling ERMR. $K \subseteq M$ is an invariant, and hence ERMR is called at least as many times as RMR in (a).



(a)

(b)

Figure 3. (a) The clauses in every vertex cut of an unsatisfiability proof must be unsatisfiable, since together they imply the empty clause (b) consequently, any assignment that satisfies $\Psi \setminus cone(c, \pi)$ (the clauses within the dashed polygon) *cannot* satisfy clauses that appear in all paths of $cone(c, \pi)$: otherwise a vertex cut can be satisfied. In this case c and $c1$ are such clauses. Hence $\Psi \setminus cone(c, \pi)$ is equisatisfiable to $\Psi \setminus cone(c, \pi) \cup \neg c \cup \neg c1$.

at least one clause in any *vertex cut* of $cone(c, \pi)$, because otherwise a satisfiable vertex cut in π would exist. Fig. 3 illustrates this property — see caption. An immediate corollary is that *all* the clauses in *some* path in $cone(c, \pi)$ (i.e., a path from c to $\bot$) must be falsified by any model h to $S \setminus cone(c, \pi)$. This implies that a clause that appears in all paths of $cone(c, \pi)$ cannot be satisfied by h.

We use this property as follows. Let $P = [c_0 = c, c_1, \ldots, c_m]$ be a path in the resolution proof starting from a candidate clause c. P is the *longest unique prefix* if it is the longest path starting at c, such that each $c_i \in P$ has only one child (that is, c participates in the derivation of one clause only). Clearly clauses in P participate in any path of $cone(c, \pi)$, and we can therefore add their negation when checking $\Psi \setminus cone(c, \pi)$. This is called *Path strengthening*. Alg. 5 shows a variant of the main algorithm in which path strengthening has been applied: each invocation of the SAT solver is carried out under the assumptions $\neg P = \{\neg c_0, \ldots, \neg c_m\}$. Before each iteration our algorithm attempts to increase the length of P by removing from the resolution proof clauses that are not backward reachable from the empty clause. Note that whenever P contains clauses which do not subsume c, path strengthening will provide more assumptions to the solver than *redundancy removal* [46, 5], which is implemented in MUSer2. Redundancy removal adds the literals of $\neg c$ (where c is the candidate clause) as assumptions when checking the satisfiability of $S \setminus c$. Observe that this is a special case of path strengthening. Hence path strengthening is expected to be more efficient than redundancy removal.

Cut falsifiability-based techniques are not immediately compliant with clause set refinement, since clause set refinement requires solving *without assumptions*. MUSer2 solves this problem for redundancy removal by applying clause set refinement only when the assumptions are not used in the proof; otherwise it skips clause set refinement. Our path strengthening algorithm applies clause set refinement when either the assumptions are not used in the proof, or the following condition holds (line 10): for a user-given constant N, the N latest iterations were UNSAT and used assumptions, i.e., line 10 was reached in the last N iterations of the loop. This implies that we did not enjoy the benefits of clause set

Algorithm 5 An improvement of Alg. 4 based on path strengthening. In line 15 the literals defined by $\{\neg c_i \mid c_i \in P\}$ are assumptions.

```
Input:   Unsatisfiability proof $\pi$ of $\Psi$.
Output:  A MUC of $\Psi$.

 1: $IsSAT := false$;
 2: $M := P := \emptyset$;
 3: while ($true$) do
 4:   if ($IsSAT$) then
 5:     $K := \{c\}$;
 6:     $M := ERMR(\Psi, c, M, K, h)$                  ▷ $h$ is the satisfying assignment
 7:   else
 8:     if proof relies on assumptions then      ▷ In first iteration the condition is false
 9:       $\pi := \pi \setminus cone(c, \pi)$;
10:       if $condition$ then                                   ▷ Heuristic. See text
11:         $\langle IsSAT, \pi \rangle := SAT(\pi)$;                  ▷ guaranteed unsat
12:   if $core(\pi) = M$ then break;
13:   Choose $c \in (core(\pi) \setminus M)$;
14:   Let $P$ be the longest unique prefix from $c$;
15:   $\langle IsSAT, \pi \rangle := SAT(\pi \setminus cone(c, \pi), \{\neg c_i \mid c_i \in P\})$;   ▷ Second parameter is a set of
    assumptions
16: return $M$;
```

refinement for N iterations, rather we progressed one clause at a time; it is possibly better to pay the price of an additional SAT call (line 11) for the benefit of clause-set refinement.

Since path strengthening is based on finding a joint prefix of the proof from the removed clause C, it is not applicable to HLMUC, since in HLMUC we remove multiple clauses ("roots") each time, which prevents a joint prefix.

4. Experimental results

We tested the effect of the nine optimizations with hundreds of industrial problems, as reported next.

4.1 HLMUC experiments

Our tool HHLMUC (for Haifa's high-level MUC) was built, as mentioned earlier, on top of Minisat 2.2. It contains the algorithm from Sect. 2 and also the technique of [42] for reducing the amount of required data in the resolution table by using a reference-counter. On top of this we implemented the optimizations that were described in the previous section, and ran all possible combinations (excluding the restrictions mentioned in optimization E, and excluding optimization H — see below a separate experiment with that optimization), on the set used in [34] (family 'lat-fmcad10' in the tables below), and additional nine families of harder abstraction-refinement benchmarks from Intel. We removed from the benchmark set instances that could not be solved by any of the configurations in the given time-out of one hour. This left us with 144 benchmarks, all of which are from the two application domains that were described in the introduction. This set constitute Intel's contribution to the benchmarks repository that was used in the SAT competition dedicated to this problem. The average number of clauses per instance is 2,572,270; the average number of constraints per instance is 3804; and, finally, the average number of clauses in the high-level constraints per instance is 96568 (25.3 clauses per constraint), which is approximately 6% of the clauses. All experiments were ran on Intel® Xeon® machines with 4Ghz CPU frequency and 32Gb of memory.

Table 2 shows run time results for selected configurations.[6.] The second column ("Base") refers to our starting point, namely an implementation of Alg. 2. One may observe that the best result is achieved when combining the first six optimizations, whereas the seventh slightly increases the overall run-time.

We also compared our results to assumptions-based minimization. We tried two methods. In the simple method, a constraint is added to the MUC (line 5 in Alg. 2) by setting its associated selector variable to true; in the improved method the same effect is achieved by adding a unit clause asserting this literal to TRUE. Similarly, in the simple method an environment assumption is removed from the formula (line 9 in Alg. 2) by setting its associated selector to FALSE; In the improved method the same effect is achieved by adding a unit clause asserting this literal to FALSE. The improved method is better empirically apparently because the unit clause invokes a simplification step in decision level 0, which removes the selector variable and erases some clauses. The results we witnessed with the two methods appear in the last two columns of the table. Overall the combination of optimizations achieve a reduction of 55% in run time comparing to our starting point, and a reduction of 28% comparing to the assumptions-based method.

All the presented methods can be affected by the order in which constraints are removed in line 12 of Alg. 2. We therefore tried three different arbitrary removal orders in each case. Empirically this hardly had an effect on the average run-time when using the resolution-based methods, whereas it had some effect when using the assumption-based methods. The table below represents the best overall run times among the different orders we tried (i.e., we present the results that together have the minimum run-time). Regarding the size of the resulting core, the different arbitrary orders had inconsistent effect, as expected, but the order referred to in optimization G had a non-negligible positive effect on the size of the core, as will be shown momentarily.

Next, in Table 3, we consider the size of the resulting HLMUC. The configuration that achieves the best run-time (A–F) achieves the second smallest HLMUC, whereas the second best configuration in terms of run time (A–G) achieves the smallest core. If a solver timed-out in our experiments, we considered its latest computed core, i.e., the set $M \cup HLC$. If a solver did not finish even the first iteration, then we considered the entire set of clauses in HLC as its achieved core. This policy, which reflects the way such cores are used, explains the different results of strategies that are supposed to be equivalent with respect to the size of the core. For example, the partial-resolution proof optimization (A) does not remove more clauses than 'Base', but since the latter is generally slower, it times-out more times

[6.] The tool and the full set of results, including a comparison to MUC tools (which does not appear here) can be downloaded from [40].

Table 2. Summary of run-time results by family (144 instances all together). 'fm-d10' is the 'lat-fmcad10' family.

Bench.	Resolution-based								Assum.-based	
family	Base	A	AB	ABC	ABCE	A–E	A–F	A–G		units
latch1	2001	1604	660	465	570	575	425	423	819	798
gate1	3747	1403	705	636	620	579	490	477	856	855
latch2	9113	5915	6636	6116	5685	5656	2424	2370	8153	8043
latch3	348	293	274	274	283	275	262	200	236	236
latch4	769	529	506	457	467	455	443	379	504	521
latch5	1103	820	735	657	678	630	632	625	747	689
fm-d10	785	457	445	451	435	435	400	394	417	425
latch6	8868	5456	5329	5188	5007	5006	4948	4943	5322	5279
latch7	9956	7050	5719	5244	5094	5096	5302	5286	5688	5652
latch8	8223	7946	5673	6133	5459	5420	5127	5587	8004	5534
Total	44913	31473	26682	25621	24298	24127	**20453**	20684	30746	28032

Table 3. Summary of the size of the HLMUC by family. The 'TO' row indicates the number of time-outs.

Bench.	Resolution-based								Assum.-based	
family	Base	A	AB	ABC	ABCE	A–E	A-F	A-G		units
latch1	41	41	41	41	42	42	41	42	52	45
gate1	1143	1210	1089	568	1029	1029	870	901	618	1192
latch2	5887	2851	127	3040	2851	2851	131	129	3782	4165
latch3	168	202	202	199	211	211	208	123	140	132
latch4	236	237	248	236	238	238	237	162	177	217
latch5	224	266	266	206	206	206	220	222	222	223
fm-d10	577	456	456	489	540	540	453	454	457	450
latch6	2550	2502	2502	2490	2490	2490	2480	2480	2463	2502
latch7	2578	322	585	253	154	154	211	204	304	287
latch8	5591	615	2867	393	344	344	371	373	2887	2877
TO	8	5	3	3	2	2	2	2	6	5
Total	18995	8702	8383	7915	8105	8105	5222	**5090**	11102	12090

and hence its core count is larger. The 'TO' row contains the number of such time-outs with each configuration.

The effect of rotation (H), and a comparison to MUSer2 We recently conducted another set of experiments, this time with the 197 GMUS competition benchmarks in order to compare ourselves to the latest version of MUSer2 and also for checking the effect of optimization H (eager rotation) which we recently implemented. The experiments were

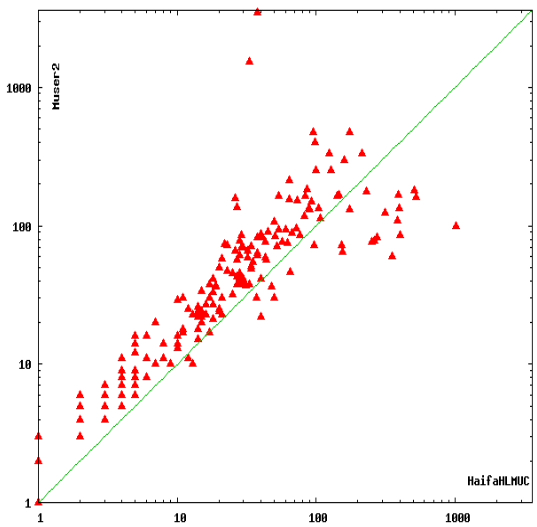


Figure 4. Comparing HaifaHLMUC to MUSer2 (Nov. 2014) with the 2011 GMUS competition benchmarks.

conducted on Intel® Xeon® CPU E5-2670 processors with 2.60GHz CPU frequency and having 64Gb of memory. The results are:

- HaifaHLMUC with optimization A – G: 28187 sec.
- HaifaHLMUC with optimization A – H: 11483 sec.
- MUSer2 (default configuration, which includes rotation): 20440 sec.

Hence the winning strategy, which is now the default of HaifaHLMUC, is A – H. A detailed comparison of HaifaHLMUC to MUSer2 can be seen in Fig. 4.

4.2 MUC experiments

We checked the impact of our algorithms when applied to the 295 instances used for the MUC track of the SAT 2011 competition. For the experiments we used machines with 32Gb of memory running Intel® Xeon® processors with 3Ghz CPU frequency. The time-out was set to 1800 sec. The implementation was done in HaifaMUC. We refer to a configuration of HaifaMUC that implements the deletion-based algorithm with incremental SAT and clause set refinement as Base. We compare our tool to the latest version of MUSer2 [7] and Minisatabb [24]. MUSer2 applies the basic deletion-based approach to MUC extraction, described in Section 2, using assumptions to keep track of dependencies. It enhances the basic deletion-based approach by rotation and redundancy removal, which we described as part of optimizations H and I. Minisatabb is an extension of MUSer2: it replaces blocks of assumptions with new variables and stores the dependencies between them. This technique is similar in essence to our optimization A (instead of storing resolution information, it stores these dependencies). In addition, Minisatabb applies clause minimization selectively,

Table 4. Total run-time in sec. and number of unsolved instances for various solvers, when applied to the 295 instances from the 2011 MUC competition, excluding 12 instances which were not solved by any of the solvers (the time-out value of 1800 sec. was added to the run-time when a memory-out occurred). Base is defined in Sect. 4.2, rot = Base+rotation, erot = Base+eager rotation. A, B, C, and D correspond to the optimizations defined in Sect. 3. '2' in AB2CD means that the optimization was invoked after the 2nd satisfiable result. 'rr' refers to redundancy removal combined with clause set refinement using MUSer2's scheme, described in Sect. 3. 'ps20' means that path strengthening with $N = 20$ was applied as described in Sect. 3.

	Base	rot	erot	erot_	erot_	erot_	erot_	**erot_**	**erot_**
				AD	ABD	AB2D	AB2CD	**AB2CD_rr**	**AB2CD_ps20**
Time	93931	48018	44335	36295	37798	32968	32918	30800	**27263**
Unsolved	30	12	10	8	13	8	8	6	**4**

	MUSer2	Minisatabb
Time	59502	40485
Unsolved	17	8

which is similar to our optimization B. Extended experimental data is available from the second author's home page.

Table. 4 summarizes the main results. Several observations are in order: 1) rotation is very useful; 2) eager rotation is effective; 3) optimizations A and D are useful, while optimization B is beneficial only if delayed until the second satisfiable iteration (2 being the optimal value, based on experiments); 4) path strengthening (with N=20, 20 being the optimal value experimentally) is more beneficial than redundancy removal, and finally 5) HaifaMUC, enhanced by all our algorithms, is 2.18x faster than MUSer2 and solves 13 more instances, and is 48% faster than Minisatabb and solves 4 more instances. HaifaMUC is faster than Minisatabb on 196 instances, while Minisatabb is faster than HaifaMUC on 15 instances. Fig. 5 compares HaifaMUC to Minisatabb and Fig. 6 shows a cactus plot comparing Base, MUSer2, Minisatabb and the new best configuration of HaifaMUC.

5. Summary and future work

We presented a basic deletion-based method for finding high-level unsat cores. Earlier methods were based on retrieving first a minimal core, and then deriving from it a high-level core, which means that it was not necessarily minimal. We also presented nine optimizations to MUC- and HLMUC-extraction, although not all apply to both goals. Some of these optimizations bias the search itself towards a minimal core. These are the main techniques underlying our tools HaifaMUC and HaifaHLMUC, which are currently the fastest of their kind.

A straight-forward direction for future research is to migrate some of the suggested optimizations to the assumptions-based approach. Related SAT problems may also benefit from these methods. First, it is possible that general SAT solving can be improved with some combination of optimizations E and F. Second, the same techniques can potentially expedite other methods in which the SAT component needs to extract only partial information from the resolution proof, like interpolation-based model checking [32]. In interpolation only a

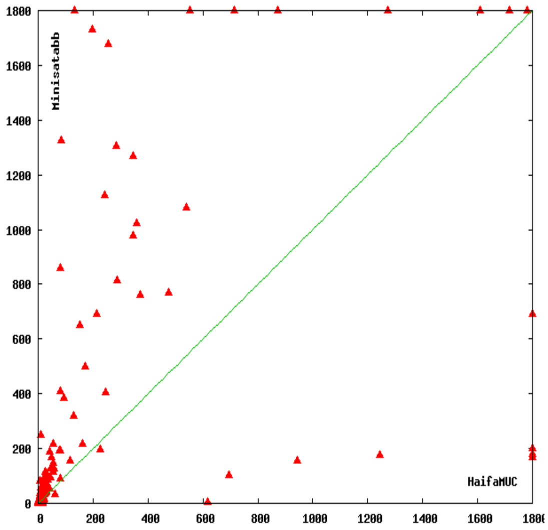


Figure 5. Direct comparison of the new best configuration of HaifaMUC erot_AB2CD_ps20 (X-Axis) and Minisatabb (Y-Axis).

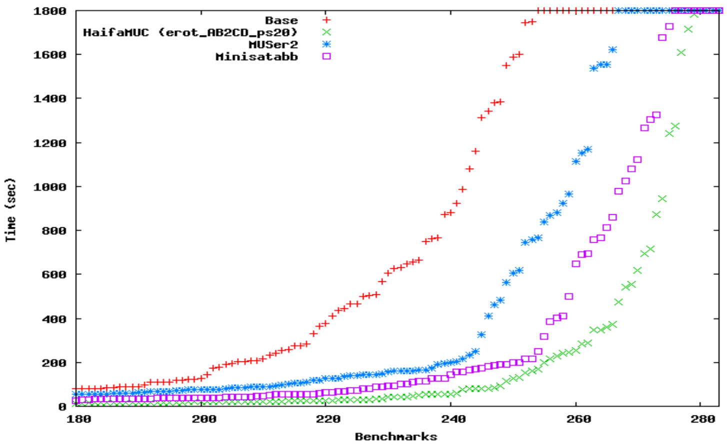


Figure 6. Comparison of Base, MUSer2, Minisatabb, and the new best configuration of HaifaMUC erot_AB2CD_ps20. The graph shows the number of solved instances (X-Axis) per time-out in seconds (Y-Axis) for each solver.

small part of the proof is necessary in order to generate the interpolant, so it should be useful to explore possibilities to minimize that part and decrease the overall run time with variants of the methods suggested here.

References

[1] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in SAT. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, **5330** of *LNCS*, pages 16–30. Springer, 2008.

[2] R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *IJCAI*, pages 276–281, 1993.

[3] P. Beame., H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, **22**:319–351, 2004.

[4] Anton Belov, Huan Chen, Alan Mishchenko, and Joao Marques-Silva. Core minimization in SAT-based abstraction. In Enrico Macii, editor, *DATE*, pages 1411–1416. EDA Consortium San Jose, CA, USA / ACM DL, 2013.

[5] Anton Belov, Inês Lynce, and João Marques-Silva. Towards efficient MUS extraction. *AI Commun.*, **25**(2):97–116, 2012.

[6] Anton Belov and João Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *FMCAD*, pages 37–40, 2011.

[7] Anton Belov and João Marques-Silva. MUSer2: An efficient MUS extractor. *J. on Satisfiability, Boolean Modeling and Computation (JSAT)*, **8**(1/2):123–128, 2012.

[8] Armin Biere. PicoSAT essentials. *J. on Satisfiability, Boolean Modeling and Computation (JSAT)*, **4**(2-4):75–97, 2008.

[9] Renato Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Appl. Math.*, **130**(2):85–100, 2003.

[10] Huan Chen and Joao Marques-Silva. Improvements to satisfiability-based boolean function bi-decomposition. In Salvador Mir, Chi-Ying Tsui, Ricardo Reis, and Oliver C. S. Choy, editors, *VLSI-SoC (Selected Papers)*, **379** of *IFIP Advances in Information and Communication Technology*, pages 52–72. Springer, 2011.

[11] John W. Chinneck and Erik W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *INFORMS Journal on Computing*, **3**(2):157–168, 1991.

[12] Orly Cohen, Moran Gordon, Michael Lifshits, Alexander Nadel, and Vadim Ryvchin. Designers work less with quality formal equivalence checking. In *Design and Verification Conference (DVCon)*, 2010.

[13] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 36–41, 2006.

[14] Christian Desrosiers, Philippe Galinier, Alain Hertz, and Sandrine Paroz. Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems. *J. Comb. Optim.*, **18**(2):124–150, 2009.

[15] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing (SAT)*, **2919** of *LNCS*, pages 502–518. Springer, 2003.

[16] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *ENTCS*, **89**(4), 2003.

[17] Roman Gershman, Maya Koifman, and Ofer Strichman. Deriving small unsatisfiable cores with dominators. In *Proc.* 18^{th} *Intl. Conference on Computer Aided Verification (CAV'06)*, number 4144 in LNCS, pages 109–122, 2006.

[18] Roman Gershman, Maya Koifman, and Ofer Strichman. An approach for extracting a small unsatisfiable core. *J. on Formal Methods in System Design (FMSD)*, pages 1 – 27, 2008.

[19] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, pages 886–891, 2003.

[20] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Using local search to find MSSes and MUSes. *European Journal of Operational Research*, **199**(3):640–646, 2009.

[21] A. Gupta, M. K. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *ICCAD*, pages 416–423, 2003.

[22] Anubhav Gupta. *Learning Abstractions for Model Checking.* PhD thesis, Carnegie Mellon University, 2006.

[23] Zurab Khasidashvili, Daher Kaiss, and Doron Bustan. A compositional theory for post-reboot observational equivalence checking of hardware. In *FMCAD*, pages 136–143, 2009.

[24] Jean-Marie Lagniez and Armin Biere. Factoring out assumptions to speed up MUS extraction. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing (SAT)*, **7962** of *LNCS*, pages 276–292. Springer, 2013.

[25] Ruei-Rung Lee, Jie-Hong Roland Jiang, and Wei-Lun Hung. Bi-decomposing large boolean functions via interpolation and satisfiability solving. In Limor Fix, editor, *DAC*, pages 636–641. ACM, 2008.

[26] Mark H. Liffiton, Maher N. Mneimneh, Inês Lynce, Zaher S. Andraus, João Marques-Silva, and Karem A. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas. *Constraints*, **14**(4):415–442, 2009.

[27] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, **40**(1):1–33, 2008.

[28] I. Lynce and J. Marques-Silva. On computing minimum unsatisfiable cores. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 305–310, 2004.

[29] Joao Marques-Silva, Mikolás Janota, and Anton Belov. Minimal sets over monotone predicates in boolean formulae. In Natasha Sharygina and Helmut Veith, editors, *CAV*, **8044** of *LNCS*, pages 592–607. Springer, 2013.

[30] Joao Marques-Silva and Ines Lynce. On improving MUS extraction algorithms. In *Theory and Applications of Satisfiability Testing (SAT)*, number 6695 in LNCS, pages 159–173, 2011.

[31] K. McMillan and N. Amla. Automatic abstraction without counterexamples. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, **2619** of *LNCS*, 2003.

[32] K.L. McMillan. Interpolation and SAT-based model checking. In Jr. Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification (CAV)*, LNCS, Jul 2003.

[33] Alexander Nadel. *Understanding and Improving a Modern SAT Solver*. PhD thesis, Tel Aviv University, Tel Aviv, Israel, August 2009.

[34] Alexander Nadel. Boosting minimal unsatisfiable core extraction. In Roderick Bloem and Natasha Sharygina, editors, *FMCAD*, 2010.

[35] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient mus extraction with resolution. In *FMCAD*, pages 197–200. IEEE, 2013.

[36] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. Amuse: a minimally-unsatisfiable subformula extractor. In *DAC '04*, pages 518–523, 2004.

[37] Christos H. Papadimitriou and David Wolfe. The complexity of facets resolved. *J. Comput. Syst. Sci.*, **37**(1):2–13, 1988.

[38] Cédric Piette, Youssef Hamadi, and Lakhdar Sais. Efficient combination of decision procedures for MUS computation. In Silvio Ghilardi and Roberto Sebastiani, editors, *FroCos*, **5749** of *LNCS*, pages 335–349. Springer, 2009.

[39] Vadim Ryvchin. *Core algorithms for SAT and SAT-related problems*. PhD thesis, Technion, IE, Haifa, Israel, 2014.

[40] Vadim Ryvchin and Ofer Strichman. Benchmarks + results: http://ie.technion.ac.il/~ofers/sat11.html.

[41] Vadim Ryvchin and Ofer Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *Theory and Applications of Satisfiability Testing (SAT)*, number 6695 in LNCS, pages 174–187, 2011.

[42] Ohad Shacham and Karen Yorav. On-the-fly resolve trace minimization. In *DAC*, pages 594–599, 2007.

[43] João P. Marques Silva. Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *ISMVL'10*, pages 9–14. IEEE Computer Society, 2010.

[44] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing (SAT)*, **5584** of *LNCS*, pages 237–243. Springer, 2009.

[45] Ofer Strichman. Pruning techniques for the SAT-based bounded model checking problem. In Tiziana Margaria and Thomas F. Melham, editors, *CHARME*, **2144** of *LNCS*, pages 58–70. Springer, 2001.

[46] Hans van Maaren and Siert Wieringa. Finding guaranteed muses fast. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing (SAT)*, **4996** of *LNCS*, pages 291–304. Springer, 2008.

[47] Jesse Whittemore, Joonyoung Kim, and Karem A. Sakallah. SATIRE: A new incremental satisfiability engine. In *DAC*, pages 542–545. ACM, 2001.

[48] Siert Wieringa. Understanding, improving and parallelizing MUS finding using model rotation. In Michela Milano, editor, *CP'12*, **7514** of *LNCS*, pages 672–687. Springer, 2012.

[49] Jianmin Zhang, Sikun Li, and ShengYu Shen. Extracting minimum unsatisfiable cores with a greedy genetic algorithm. In Abdul Sattar and Byeong Ho Kang, editors, *Australian Conference on Artificial Intelligence*, **4304** of *LNCS*, pages 847–856. Springer, 2006.

[50] Jianmin Zhang, ShengYu Shen, and Sikun Li. Tracking unsatisfiable subformulas from reduced refutation proof. *JSW*, **4**(1):42–49, 2009.

[51] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Theory and Applications of Satisfiability Testing (SAT)*, 2003.