Faster Extraction of High-Level Minimal Unsatisfiable Cores

Vadim Ryvchin [1,2] and Ofer Strichman [1]

[1] Information Systems Engineering, IE, Technion, Haifa, Israel**
[2] Design Technology Solutions Group, Intel Corporation, Haifa, Israel
rvadim@tx.technion.ac.il ofers@ie.technion.ac.il

Abstract. Various verification techniques are based on SAT's capability to identify a small, or even minimal, unsatisfiable core in case the formula is unsatisfiable, i.e., a small subset of the clauses that are unsatisfiable regardless of the rest of the formula. In most cases it is not the core itself that is being used, rather it is processed further in order to check which clauses from a preknown set of *Interesting Constraints* (where each constraint is modeled with a conjunction of clauses) participate in the proof. The problem of minimizing the participation of interesting constraints was recently coined *high-level* minimal unsatisfiable core by Nadel [15]. Two prominent examples of verification techniques that need such small cores are 1) abstraction-refinement model-checking techniques, which use the core in order to identify the state variables that will be used for refinement (smaller number of such variables in the core implies that more state variables can be replaced with free inputs in the abstract model), and 2) assumption minimization, where the goal is to minimize the usage of environment assumptions in the proof, because these assumptions have to be proved separately. We propose seven improvements to the recent solution given in [15], which together result in an overall reduction of 55% in run time and 73% in the size of the resulting core, based on our experiments with hundreds of industrial test cases. The optimized procedure is also better empirically than the assumptions-based minimization technique.

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 [8]. Finding a *minimal* UC (a UC such that the removal of any one of its clauses makes the formula satisfiable) is D^P-complete [17][1]. There are many works in the literature on extracting minimum [8, 11], minimal [16, 3, 12, 21], or just small cores [22, 6, 4] — see [15] for an extensive survey.

** Currently on sabbatical at the Software Engineering Institute, Pittsburgh, PA, USA

[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.

There are many uses to the core in SAT-based verification, typically related to abstraction or decomposition. In most cases, however, it is not the core C itself that is being used, rather C is processed further in order to check which *Interesting Constraints* participate in the proof, where which constraints are interesting 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 sets of clauses $IC = \{R_1 \ldots R_m\}$, where each R_i is a set of clauses that together encode an interesting constraint. The goal is thus to minimize the number of constraints in IC that have a non-empty intersection with C. This problem was first mentioned in [12] and recently coined *the high-level minimal unsatisfiable core* problem by Nadel [15], who observed that in his experiments with industrial problems the number of clauses that belong to interesting constraints is on average about 5% of the clause database. In fact in the verification group in Intel high-level cores are the only type of cores that are being computed, and we are not aware of any use of the general core in the EDA industry.

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 [14, 9]. 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 IC. 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., [10]), 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 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 IC is a set of clauses that encode an

environment assumption. Here too the verification process attempts to minimize the high-level core in order to minimize the number of environment assumptions that should be verified.

We will address the question of how to minimize the core in the next section. A problem which is mostly orthogonal to minimization is how to make the SAT solver emit a core once it determines that a formula is unsatisfiable. There are two well-known approaches to solve this problem:

– **Resolution-based.** The first approach is based on the ability of many modern SAT solvers to produce a resolution proof in case the formula is unsatisfiable. The solver traverses the proof backwards from the empty clause, and reports the clauses at the leaves as the core [22, 7]. This core is then intersected with the sets of clauses in IC in order to find a high-level core.
– **Assumptions-based.** A second approach is based on the *assumptions* technique, which was first implemented in an early version of Minisat [5]. Assumptions are literals that are assigned TRUE (as decisions) before any other decision. If constraint propagation leads to flipping the assignment of one of the assumptions to FALSE, it means that with these assumptions the formula is unsatisfiable. Minisat is capable of identifying which assumptions led to this conflict, which is exactly what is needed for extracting a high-level core. This can be done with *clause selectors* as follows: Let R_i be constraint in IC and let $\{c_1, \ldots, c_n\}$ be the clauses that encode it. To each clause in this set we add the literal $\neg l_i$, where l_i is a new variable. Then we add l_i to the set of assumptions. Hence setting l_i to TRUE activates this constraint, and setting it to FALSE deactivates it.
  The process of extracting the set of assumptions that led to a conflict is computationally easy. Let C be the clause that forces an assumption to its opposite value. Minisat resolves C with all its predecessors in the implication graph until a clause is generated which contains only negation of assumption literals. The negation of this clause is a conjunction of the assumptions that led to the conflict, also known as the *relevant assumptions.* The relevant assumptions constitute a high-level core.

The assumptions technique generates larger conflict clauses owing to the new selector variables, which may become significant if there are many assumptions [15, 1]. The alternative of activating and deactivating constraints with unit clauses is more economic, as it simplifies and removes clauses. On the other hand, the assumptions technique does not consume memory for saving the proof, nor does it consume time to extract the core. Another difference between these two approaches, which turns out to be very important in our context, is related to clause minimization [2, 20], which is a technique for shrinking conflict clauses. Whereas in resolution-based core extraction minimization of a clause may pull into the proof additional constraints, this does not happen in the assumptions-based approach. We will describe this issue in more detail in Sect. 4. The experiments in [15] showed that the assumptions-based method is on average faster than the resolution-based method, and produces slightly smaller cores. In the experiments we conducted (on a larger set of benchmarks) we witnessed similar results.

In this article we study seven improvements to the resolution-based high-level MUC problem. With these techniques, which we implemented on top of MiniSat-2.2 and ran over hundreds of industrial examples from Intel, we are able to show a 55% reduction in run time comparing to the techniques in [15], and a 28% improvement comparing to the assumptions-based technique. The configuration that achieves these improvements also reduces the core by 73% and 57%, respectively. More details on our experiments can be found in Sect. 4.

Since we take [15] as the starting point of our optimizations, we begin in the next section by describing it in some detail.

2 Resolution-based high-level core minimization

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. [19]. 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. In the experiments that were reported in [19], 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. We implemented the high-level core minimization algorithm of [15], which appears in Pseudo-code in Alg. 1. The input to this algorithm is a set of interesting constraints $IC = \{R_1 \dots R_m\}$, each of which is a set (or a conjunction, depending on the context) of clauses, and a formula Ω, which is called the *remainder*. The formula $\Psi = \bigwedge_{j=1}^{m} R_j \wedge \Omega$ is assumed to be unsatisfiable, and the proof is available at the beginning of the algorithm. We denote the initial core by *initial_core*. The output of the algorithm is a high-level minimal unsatisfiable core with respect to IC and Ω, i.e., a subset $IC' \subseteq IC$ such that $\Psi' = \bigwedge_{R_j \in IC'} R_j \wedge \Omega$ is unsatisfiable, and no constraint can be removed of IC' without making Ψ' satisfiable.

The algorithm is rather self-explanatory, so we will be brief in describing it. In line 1 any constraint R_i that none of its clauses participated in the proof is removed together with its cone, i.e., all the clauses that were derived (transitively) from R_i clauses. The next line defines the set of candidate indices for the core, which is initiated to the indices of the constraints in IC that were not removed in the previous step. From here on the algorithm attempts to remove elements of this set. In each iteration of the loop, it removes a constraint R_k together with its cone and checks for satisfiability. If the formula is satisfiable, then R_k with its cone is returned to the formula, and R_k is added to the solution set *muc*. Otherwise, the unsatisfiability proof is checked in order to remove any constraint R_i, together with its cone, that did not participate in the proof.

Algorithm 1 Resolution-based high-level MUC extraction (Based on Alg. 2 in [15])

Input: Unsatisfiable formula of the form $\Psi = \bigwedge_{R_j \in IC} R_j \wedge \Omega$.
Output: A high-level MUC with respect to IC and Ω.

```
1: Remove any $R_i$ together with its cone if it is not reachable from the empty clause;
2: $muc\_cands := \{R_i \mid R_i \cap initial\_core \neq \emptyset\}$;    ▷ MUC Candidates
3: $muc := \{\}$;
4: while $muc\_cands$ is non-empty do
5:     $R_k :=$ a member of $muc\_cands$;
6:     Check satisfiability of the formula without $R_k$ and its cone;
7:     if satisfiable then
8:         return $R_k$ and its cone to the formula;
9:         $muc := muc \cup \{R_k\}$;
10:    else
11:        for $R_i \in muc\_cands$ do
12:            if $R_i \cap core = \emptyset$ then    ▷ $core$ is the unsat core of the proof
13:                Remove $R_i$ and its cone;
14:                $muc\_cands := muc\_cands \setminus \{R_i\}$;
15: return $muc$;
```

It is interesting to note that this algorithm is tailored for *high-level* core minimization, and not for general core minimization. The difference is evident by observing that the whole set of clauses associated with a constraint R_i is removed, together with their joint core. Had the object of minimization been the whole core, we would rather remove all clauses that did not participate in the proof, even if other clauses that share the same constraint *do* participate in the proof. For example, if $R_i = \{c_1, c_2\}$, and only c_1 participate in the proof, Alg. 1 retains both c_1 and c_2, because removing c_2 does not reduce the size of the high-level core, whereas it may assist in consecutive iterations. Furthermore, retaining c_2 is needed in order to guarantee minimality. Without it we may miss the fact that some other constraint can be removed.

3 Optimizations

In this section we describe seven low-level optimizations to the basic algorithm that was presented in the previous section. We will use the following terminology: a clause is an *IC-clause* if it either belongs to one of the initial constraints in *IC* or is a descendant of such a clause in the resolution DAG. Other clauses are called *remainder* clauses. We say that a literal is *IC-implied* if it is implied by an IC-clause or just *implied* otherwise.

A: Maintaining partial resolution proofs. In this optimization we maintain only clauses in the cone of IC-clauses in the resolution table, and the links between them. That is, we save an IC-clause, and the parents and children that are also IC-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 clauses that are in the cone of an IC (recall that in line 13 of Alg. 1 IC-clauses are removed together with their cones), and, more importantly, allows to activate a certain simplification (see below) for remainder clauses, which is normally turned off when running Alg. 1.

The simplification in point is applied in decision level 0, owing to constants. 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 when running Alg. 1. For example, if (x) is an IC-clause associated with constraint R_1, then we cannot just remove clauses with x from the formula, since we might decide at line 13 to remove R_1, which will force us to retrieve these clauses. Empirically it is better to retain such clauses rather than keeping them in a file and then retrieving them. The same issue occurs when removing the negation of x from clauses: here too, we will need to retrieve the original clauses once R_1 is removed. One of the advantages of this optimization, therefore, is that we can turn back on this simplification for the remainder clauses.

B: Selective clause minimization. Clause minimization [2, 20] 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:

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

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-IC-clause C into a shorter IC-clause C'. This can happen if the minimization process uses an IC-clause: in that case C' has to be marked as an IC-clause as well. Furthermore, it can turn an IC-clause C that depends on a certain set of interesting constraints, into a shorter IC-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 IC-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 IC-constraints. Ideally we should check whether using a certain IC-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 IC-clauses have as literals all the selector variables that correspond to constraints that were used in deriving that clause. For example, let R_1, R_2 be two constraints with associated selector variables l_1, l_2 respectively. If R_1 and R_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 IC-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 *IC*-clauses. In this optimization we control the BCP order. We first run BCP over non-IC-clauses until completion. If there is no conflict, we propagate a single implication due to an IC-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 IC-clause.

D: Reclassifying *IC*-clauses. When we discover that some IC-constraint R must be in the MUC (line 8 in Alg. 1), 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 R 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 R's clauses are added back as is, with their marking as IC-clauses. By adding them back as remainder clauses, we enable more simplifications, as described in the case of optimization **A**.

E: Selective learning of *IC*-clauses. When detecting a conflict, the learned clause may be an IC-clause. If all else is equal, such a clause is less preferable than a remainder clause, as it may increase the high-level core, 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 IC-clause; in other cases we simply revert to learning the IC-clause. Learning the remainder clause is done by reanalyzing the conflict graph *as if the IC-implications were decisions*. This optimization is only ran in conjunction with optimizations **B** and **C** above, for reasons that we will soon clarify. Alg. 2 describes the procedure for learning this clause.

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-IC-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 IC-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 IC-clause owing to clause minimization. This is not essential for correctness, however: we could also have just compared this smaller IC-clause to the original one and choose between the two, but our experience is that it is better to give

Algorithm 2 An algorithm that attempts to find a remainder conflict clause by reanalyzing the conflict graph as if the IC-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 $IC$-clause then return NULL.
 2. Search an $IC$-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 $C$.
```

priority to minimizing the number of IC-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 IC-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 IC-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 in neither necessary or 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. Since the solver cannot enter a conflict state that leads to learning an existing clause, we are guaranteed not to enter an infinite loop.

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 IC-clause — progress to level 4 and decides $\neg l_1$. □

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 IC-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 [18], shows that this heuristic improves the run time in most instances, and that it improves the search itself and not only reduces constants, as is evident by the fact that it reduces the number of conflicts. It seems that the reason for the success of this heuristic is related to the fact that with 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.

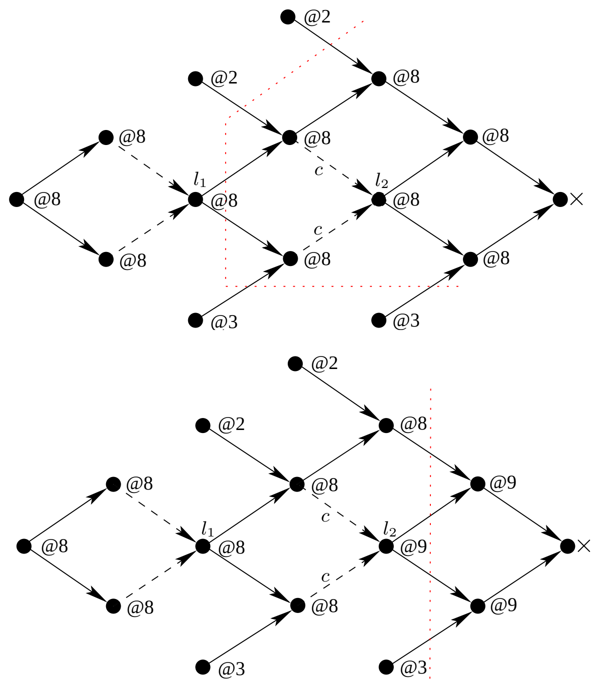

Fig. 1. In these conflict graphs, dashed arrows denote IC-implications, and the dotted lines denote 1-UIP cuts. In the top drawing, where such implications are referred to as any other implications, the learned 1-UIP clause must be marked as an IC-clause, since it is resolved from the IC-clause c. We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

G: A removal strategy. Recall that in line 5 of Alg. 1 constraints are removed in an arbitrary order. We suggest a simple greedy heuristic instead: remove the 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 IC-clauses.

4 Experimental results

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 [19] 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**), on the set used in [15] (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 will be used in the upcoming 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 interesting clauses 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 4 shows run time results for selected configurations.[2] The second column ("Full") refers to our starting point as explained above. 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 both a simple scheme, and the improvement suggested in [15]. In the simple

[2] The full set of results, including a comparison to MUC tools (which does not appear here) can be downloaded from [18]. The same web page includes a link to our tools.

scheme, a constraint is added to the MUC (line 8 in Alg.1) 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 scheme an environment assumption is removed from the formula (line 13 in Alg.1) 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 5. 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.

Benchmark	Resolution-based								Assumptions-based	
family	Full	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
lat-fmcad10	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 1. Summary of run-time results by family (144 instances all together).

Next, we consider the size of the resulting high-level MUC. The configuration that achieves the best run-time (A–F) achieves the second smallest high-level core, 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 $muc \cup muc\_cands$. If a solver did not finish even the first iteration, then we considered the entire set of clauses in IC 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 'Full', but since the latter is generally slower, it times-out more times and hence its core count is larger. The 'TO' row contains the number of such time-outs with each configuration.

Benchmark	Resolution-based								Assumptions-based	
family	Full	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
lat-fmcad10	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

Table 2. Summary of the size of the high-level core by family. The 'TO' row indicates the number of time-outs.

5 Summary and future work

The recently introduced problem of finding a *high-level* minimal unsatisfiable core has various applications in the industry. Until [15] the standard practice was to minimize the core itself, and only then to find the interesting part of it. Our experiments show that this approach cannot compete with a solver that focuses on the high-level core. In this article we introduced seven techniques that reduce both the run time and the resulting high-level core.

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 [13]. In interpolation only a small part of the proof is necessary in order to generate the interpolant, and we want to explore possibilities to minimize that part and decrease the overall run time with variants of the methods suggested here.

References

1. R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in SAT. In *LPAR*, pages 16–30, 2008.
2. 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.
3. N. Dershowitz, Z. Hanna, and A. Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *SAT*, pages 36–41, 2006.
4. C. Desrosiers, P. Galinier, A. Hertz, and S. Paroz. Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems. *J. Comb. Optim.*, 18(2):124–150, 2009.
5. N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.
6. R. Gershman, M. Koifman, and O. Strichman. An approach for extracting a small unsatisfiable core. *J. on Formal Methods in System Design*, pages 1 – 27, 2008.
7. E. Goldberg and Y. 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.
8. A. Gupta. *Learning Abstractions for Model Checking*. PhD thesis, Carnegie Mellon University, 2006.
9. 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.
10. Z. Khasidashvili, D. Kaiss, and D. Bustan. A compositional theory for post-reboot observational equivalence checking of hardware. In *FMCAD*, pages 136–143, 2009.
11. M. H. Liffiton, M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. Marques-Silva, and K. A. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas. *Constraints*, 14(4):415–442, 2009.
12. M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.
13. K. McMillan. Interpolation and sat-based model checking. In J. Warren A. Hunt and F. Somenzi, editors, *cav03*, Lect. Notes in Comp. Sci., Jul 2003.
14. K. McMillan and N. Amla. Automatic abstraction without counterexamples. In H. Garavel and J. Hatcliff, editors, *TACAS'03*, volume 2619 of *Lect. Notes in Comp. Sci.*, 2003.
15. A. Nadel. Boosting minimal unsatisfiable core extraction. In R. Bloem and N. Sharygina, editors, *FMCAD*, 2010.
16. Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. Amuse: a minimally-unsatisfiable subformula extractor. In *DAC '04*, pages 518–523, 2004.
17. C. H. Papadimitriou and D. Wolfe. The complexity of facets resolved. *J. Comput. Syst. Sci.*, 37(1):2–13, 1988.
18. V. Ryvchin. Benchmarks + results: http://ie.technion.ac.il/∼ofers/sat11.html.
19. O. Shacham and K. Yorav. On-the-fly resolve trace minimization. In *DAC*, pages 594–599, 2007.
20. N. Sörensson and A. Biere. Minimizing learned clauses. In O. Kullmann, editor, *SAT*, volume 5584 of *LNCS*, pages 237–243. Springer, 2009.
21. H. van Maaren and S. Wieringa. Finding guaranteed muses fast. In *SAT*, pages 291–304, 2008.
22. L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *In Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003), S. Margherita Ligure*, 2003.