Finding Guaranteed MUSes Fast

Hans van Maaren[1] and Siert Wieringa[2,⋆]

[1] Delft University of Technology
Faculty of EWI
Mekelweg 4, 2628 CD, Delft, The Netherlands
`h.vanmaaren@tudelft.nl`
[2] Helsinki University of Technology (TKK)
Department of Information and Computer Science
P.O. Box 5400, FI-02015 TKK, Finland
`Siert.Wieringa@tkk.fi`

Abstract. We introduce an algorithm for finding a *minimal unsatisfiable subset* (MUS) of a CNF formula. We have implemented and evaluated the algorithm and found that its performance is very competitive on a wide range of benchmarks, including both formulas that are close to minimal unsatisfiable and formulas containing MUSes that are only a small fraction of the formula size.

In our simple but effective algorithm we associate assignments with clauses. The notion of *associated assignment* has emerged from our work on a Brouwer's fixed point approximation algorithm applied to satisfiability. There, clauses are regarded to be entities that order the set of assignments and that can select an assignment to be associated with them, resulting in a *Pareto optimal agreement.*

In this presentation we abandon all terminology from this theory which is superfluous with respect to the recent objective and make the paper self contained.

1 Introduction

Solvers for instances of the Boolean satisfiability problem, so called SAT solvers, have found their way into numerous applications including *electronic design automation* (EDA) [1], formal verification [2,3] and artificial intelligence [4]. In many of those applications we would like to have an explanation of the *cause* of unsatisfiability in case a formula is unsatisfiable. For example if an FPGA routing problem is translated to a Boolean formula a satisfying assignment corresponds to a valid routing, and unsatisfiability means no such routing exists [1]. In the latter case the user might want to know which part of the design caused the unroutability.

An *unsatisfiable subset* or *core* of an unsatisfiable formula is a subset of clauses from that formula the conjunction of which is unsatisfiable. A *minimal unsatisfiable subset* (MUS) is an unsatisfiable subset that becomes satisfiable if any of its

⋆ Supported by the Academy of Finland project #112016.

clauses is removed. There can be multiple MUSes in one formula. An unsatisfiable subset might help to understand at least one cause of a formula's unsatisfiability as the clauses that have been left out were not necessary to maintain unsatisfiability. Multiple algorithms have been presented for finding unsatisfiable cores that are not guaranteed to be MUSes (i.e. [5,6,7]). In this paper we present an algorithm for finding a MUS in an unsatisfiable formula. Our algorithm does not guarantee finding the minimum unsatisfiable core, which is the MUS with the least number of clauses [8].

In [6] a nice example of a small unsatisfiable FPGA routing problem can be found. It shows the use of finding multiple MUSes and is used to argue that the minimum unsatisfiable core is not necessarily the most useful core for diagnostic purposes. Other work has focused on algorithms for finding all MUSes [9,10] or the minimum unsatisfiable core [8]. Even if finding all MUSes is not feasible those algorithms might be very useful to find one or multiple MUSes under some time constraint.

Important notions for our algorithm emerged from our work [11] on applying a Brouwer's fixed point approximation algorithm [12] to satisfiability. In that work a clause is regarded to be an entity that can select an assignment from the set of all 2^n possible assignments on which it imposes a complete ordering in which all satisfying assignments are preferred over all unsatisfying assignments. Unsatisfiability is represented by the possibility to find a subset, or *coalition*, of clauses that form a *Pareto optimal agreement*. In such an agreement all clauses have chosen a unique assignment that does not satisfy themselves, while they would all prefer the chosen assignment of all other clauses in the coalition and there is no single assignment that all those clauses prefer over their own choice. The existence of this agreement proves that the preferences of the clauses are contradictionary from which the inconsistency of the formula follows. As the clauses in the coalition prove unsatisfiability of the formula they form an unsatisfiable subset. We implemented an algorithm for finding unsatisfiable subsets using this theory [11], but as it remains far from competitive we let go of most of this background and present a simple and efficient algorithm.

2 Extracting a MUS

We will represent a CNF formula $\mathcal{F}$ by a *sequence* of clauses. Furthermore a subscript will denote an element index in a sequence, so clause C_i from $\mathcal{F} = \langle C_1, C_2, ..., C_m \rangle$ is the ith clause in the sequence.

The most straightforward approach to reducing a CNF formula of m clauses to a MUS is given in Algorithm 1. This algorithm requires solving m SAT problem instances to find a MUS in the unsatisfiable set of clauses $\mathcal{F}$. In case the input set $\mathcal{F}$ is a MUS itself all the SAT problem instances have $m-1$ clauses.

Given a CNF formula $\mathcal{F}$ Algorithm 2 finds a formula $\mathcal{F}' \subseteq \mathcal{F}$ such that $\mathcal{F}'$ and $\mathcal{F}$ have the same set of satisfying assignments. The conditional addition of the clauses from $\mathcal{F}$ to $\mathcal{F}'$ is performed one by one in the order of occurrence in the sequence $\mathcal{F} = \langle C_1, C_2, ..., C_m \rangle$. A clause C_i is added to $\mathcal{F}'$ iff there exists a

Algorithm 1. naiveFindMUS($\mathcal{F}$)

```
1: $\mathcal{F}' := \mathcal{F}$
2: for $i = 1$ to $|\mathcal{F}|$ do
3:     if $\mathcal{F}' \setminus \{C_i\}$ is UNSATISFIABLE then
4:         $\mathcal{F}' := \mathcal{F}' \setminus \{C_i\}$
5:     end if
6: end for
7: return $\mathcal{F}'$
```

Algorithm 2. findEquSubset($\mathcal{F}$)

```
1: $\mathcal{F}' := \langle\rangle$
2: for $i = 1$ to $|\mathcal{F}|$ do
3:     if $\neg C_i \wedge \mathcal{F}'$ is SATISFIABLE then
4:         append $C_i$ to $\mathcal{F}'$
5:     end if
6: end for
7: return $\mathcal{F}'$
```

truth assignment that is satisfying all the clauses that have already been added to $\mathcal{F}'$ while not satisfying C_i.

Lemma 1. *The formula $\mathcal{F}'$ returned by Algorithm 2 has the same set of satisfying truth assignments as the input formula $\mathcal{F}$.*

Proof. Each clause $C_i \in \mathcal{F}$ is added to the initially empty sequence $\mathcal{F}'$ if there is an assignment satisfying the clauses already added to $\mathcal{F}'$ while not satisfying the clause C_i. If no such assignment exists then either $\mathcal{F}'$ is unsatisfiable or $\mathcal{F}' \models C_i$. If $\mathcal{F}'$ is unsatisfiable $\mathcal{F}$ must be unsatisfiable and thus both formulas have zero satisfying truth assignments. If $\mathcal{F}' \models C_i$ then C_i does not restrict the number of satisfying truth assignments in $\mathcal{F}$ as it implied by, and therefore can be derived from, the subset of $\mathcal{F}$ that was already added to $\mathcal{F}'$. $\square$

Corollary 1. *If the input $\mathcal{F}$ of Algorithm 2 is unsatisfiable so is the formula $\mathcal{F}'$ it returns*

Although it is possible to use Algorithm 2 to reduce the number of clauses in satisfiable formulas in this paper we limit ourselves to its applications for unsatisfiable formulas.

Definition 1. *A critical clause of an unsatisfiable formula $\mathcal{F}$ is a clause that belongs to every unsatisfiable subset of the formula $\mathcal{F}$.*

Proposition 1. *In a MUS every clause is a critical clause.*

Note that an unsatisfiable formula does not have to contain any critical clauses. An example of a formula without critical clauses would be a formula with two completely disjoint unsatisfiable subsets. A critical clause is called a *necessary clause* in [13].

Lemma 2. *The last clause appended to the sequence forming the unsatisfiable formula* $\mathcal{F}' = \langle C'_1, C'_2, ..., C'_m \rangle$ *which is a subset of the unsatisfiable formula* $\mathcal{F}$ *returned by Algorithm 2 is critical for* $\mathcal{F}'$*.*

Proof. By construction it holds for each clause C'_i in $\mathcal{F}'$ that there exists an assignment that satisfies all the clauses C'_j in $\mathcal{F}'$ with $j < i$. So without the clause $C'_{|\mathcal{F}'|}$ there is a satisfying assignment for the $|\mathcal{F}'| - 1$ other clauses in $\mathcal{F}'$. Consequently, every subset of $\mathcal{F}'$ that does not contain clause $C'_{|\mathcal{F}'|}$ is satisfiable. □

Algorithm 3 reduces an unsatisfiable CNF formula to a MUS. It proceeds in multiple *rounds*, proving one clause critical in every round. The lines 3 to 8 in this algorithm are similar to the pseudo code of Algorithm 2 except for the addition of a sequence M to which $\mathcal{F}'$ is initialised. This sequence M consists of all clauses that have already been proven to be critical for the unsatisfiability of the MUS we are constructing.

Algorithm 3. REDUCETOMUS($\mathcal{F}$)

```
1:  $M := \langle\rangle$
2:  while $|M| < |\mathcal{F}|$ do
3:      $\mathcal{F}' := M$
4:      for $i = 1$ to $|\mathcal{F}|$ do
5:          if ($C_i$ does not appear in $M$) and ($\neg C_i \wedge \mathcal{F}'$ is SATISFIABLE) then
6:              append $C_i$ to $\mathcal{F}'$
7:          end if
8:      end for
9:      append LAST($\mathcal{F}'$) to $M$
10:     $\mathcal{F} := \mathcal{F}'$
11: end while
```

Lemma 3. *In each round of Algorithm 3 it finds a new critical clause unless the set of critical clauses is unsatisfiable.*

Proof. The first round proceeds just like Algorithm 2 would. At the end of the round the last clause is added to M as according to Lemma 2 it is critical. In every following round the sequence $\mathcal{F}'$ is initialised to contain the clauses in M, which are all clauses that have already been proven critical. If M is satisfiable at the start of a round more clauses will be added to $\mathcal{F}'$ which will cause the last clause added in that round to be a clause that was not yet in M. According to Lemma 2 it is critical and as it was not proven critical before we have found a new critical clause. If M is unsatisfiable at the start of a round then at the end of the round $\mathcal{F}'$ will still be equal to M and the algorithm will end. □

Instead of using a sequence M to keep track of the critical clauses one can also reshuffle the sequence of clauses before each round in such a way that all the clauses already proven critical precede the other clauses in the sequence. One might also want to add a preliminary exit to the for loop as soon as $\mathcal{F}'$ becomes unsatisfiable but from experimental results this did not seem to result in an overall performance gain.

Example 1. Consider the following unsatisfiable CNF formula.

$$(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1) \ . \tag{1}$$

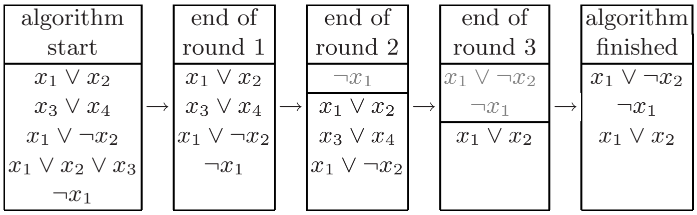


Fig. 1. Finding a MUS in Formula (1)

Each of the rectangles in Fig. 1 show the contents of the sequence $\mathcal{F}$ at some point in the execution of Algorithm 3 with the example formula as input.

- After the first round the clause $x_1 \vee x_2 \vee x_3$ is removed as there is no satisfying assignment to $\neg(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_1 \vee \neg x_2)$. This is because clause $x_1 \vee x_2 \vee x_3$ subsumes (is a superset of) clause $x_1 \vee x_2$ which has already been added to the sequence. After this round the clause $\neg x_1$ is proven to be a critical clause.
- The first clause in the rectangle in Fig. 1 holding the contents of the sequence $\mathcal{F}$ after the second round is separated from the other clauses by a horizontal line because it is a critical clause and $\mathcal{F}$ is therefore initialised to hold that clause at the start of this round. In this round no clauses are removed from the sequence. After this round clause $x_1 \vee \neg x_2$ is proven to be a critical clause.
- At the start of the third round the sequence $\mathcal{F}$ is initialised to hold the two clauses that were proven to be critical so far. Clause $x_1 \vee x_2$ can be added to the sequence as $\neg(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge \neg x_1$ is satisfiable. After adding that clause the sequence $\mathcal{F}$ becomes unsatisfiable and therefore clause $x_3 \vee x_4$ will not be added. After this round the clause $x_1 \vee x_2$ is proven to be a critical clause. As all clauses in $\mathcal{F}$ are now proven to be critical we have found a MUS. ■

We will now describe a way to find more than one critical clause per round of the algorithm. This will reduce the number of rounds the algorithm needs to find a MUS and therefore the number of SAT problem instances that need to be solved. Recall that in each round a clause C_i from $\mathcal{F}$ is only added to the sequence $\mathcal{F}'$ if there is a satisfying assignment for what is already in $\mathcal{F}'$ that is not satisfying C_i. To find multiple critical clauses in one round we store that satisfying assignment with each clause $C'_i \in \mathcal{F}'$ with $\mathcal{F}' = \langle C'_1, C'_2, ..., C'_m \rangle$ and name it the *associated assignment* of the clause C'_i. This associated assignment does not have to define a truth assignment for all variables in $\mathcal{F}$.

Definition 2. *The associated assignment of a clause $C'_i \in \mathcal{F}'$ is an assignment that satisfies all $C'_j \in \mathcal{F}'$ with $j < i$ and does not satisfy C'_i.*

Lemma 4. *If the associated assignment of a clause $C'_i \in \mathcal{F}'$ satisfies all clauses $C'_j \in \mathcal{F}'$ with $j > i$ then clause C'_i is critical for $\mathcal{F}'$.*

Proof. By definition the associated assignment of a clause $C'_i \in \mathcal{F}'$ satisfies all clauses $C'_j \in \mathcal{F}'$ with $j < i$. If it also satisfies all clauses $C'_j \in \mathcal{F}'$ with $j > i$ then it is a satisfying assignment for all C'_j in $\mathcal{F}'$ with $j \neq i$ and therefore every subset of $\mathcal{F}'$ not containing C'_i is satisfiable. □

Example 2. Let us reconsider the unsatisfiable Formula (1) and feed it to the improved version of the algorithm. Figure 2 is constructed similarly to Fig. 1 with the exception that it shows the associated assignments for the clauses in $\mathcal{F}$ after round one. Let us assume that the associated assignments define a truth assignment for all variables in Formula (1). In this example the first, third and fourth clause are proven to be critical after the first round as their associated assignments satisfy all clauses succeeding them in the sequence. At the start of the second round $\mathcal{F}'$ is initialised to contain those three clauses and therefore it becomes unsatisfiable right away. This means that clause $x_3 \vee x_4$ will not be added and we have found a MUS as all clauses in $\mathcal{F}'$ are critical.

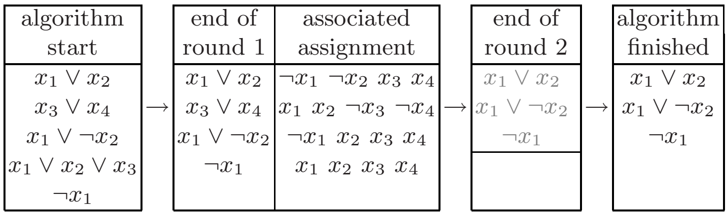


Fig. 2. Finding a MUS in Formula (1), using improved algorithm ■

The associated assignment a of a clause $C'_i \in \mathcal{F}'$ is the result of solving a SAT problem instance that only contained those literals that occurred in the clauses $C'_j \in \mathcal{F}'$ with $j \leq i$. This means that the associated assignment a of C'_i might not include a truth assignment for a literal in a clause $C'_k \in \mathcal{F}'$ with $k > i$. It is possible that none of the literals of C'_k are satisfied by the truth assignments defined in a but there is a literal in C'_k for which no truth assignment is defined in a. In an implementation of the algorithm one must either always regard such a as an unsatisfying assignment or extend a to hold a satisfying truth assignment for at least one of the literals of C'_k for which no truth assignment was previously defined in a.

Consider the case where this algorithm is given a MUS of m clauses as input for which the algorithm manages to find only one critical clause per round. In this worst case $m\frac{m+1}{2}$ SAT problem instances must be solved to prove all clauses critical, whereas the naive strategy of Algorithm 1 only requires solving m problem instances. However, first of all these SAT problem instances are varying in size from 1 to m clauses so a large number of them are easy to solve, or even trivial, while all the instances that need to be solved using the naive approach have m clauses and thus will often all be hard. Secondly, it is possible to implement all of the SAT solver calls performed in one round in one *incremental* problem, thus requiring only m incremental problems. This can be implemented easily. Adding a clause to $\mathcal{F}'$ in the pseudo code is adding a clause to the problem instance in the solver. Satisfying $\neg C_i$ without having to add it to the solvers problem instance can be guaranteed by forcing the solver to assign all literals from C_i the value false. Using the SAT solver `MiniSat` [14] such assignments can be passed as so called *assumptions*. Thirdly, our technique for finding multiple critical clauses per round will usually reduce the number of required rounds to significantly less than m as will become clear from the experimental results further on in this document.

3 Implementation

In order to obtain an efficient implementation of our algorithm we have merged it with the code of the state of the art SAT solver `MiniSat 2.0` [14] without the optional simplifier [15]. The simplifier is not used as we are interested in finding a MUS consisting of the original input clauses. It might be possible to modify the simplifier for our application but we will not discuss that here. Our MUS finder is called `MiniUnsat`. Each round is handled as one incremental SAT problem.

The problem mentioned earlier of an assignment that might not be complete for a succeeding clause is handled by extending the assignment with a satisfying truth assignment for a literal of the clause. In case there are multiple literals in the clause for which no truth assignment is defined by the assignment the literal that occurs last in the clause is chosen.

As the algorithm is greedy in the sense that it adds a clause to the subset unless it is proven to be unnecessary clause sorting has a great effect on performance in terms of speed, but also on which MUS the algorithm finds. By manually sorting the clauses before executing the program the user might give preference to one clause over the other and can thereby influence which MUS is found.

Two simple optional automatic clause sorting methods are also implemented. The first automatic sorting method, *sort by weight*, sorts clauses by the sum of the number of occurrences in the formula of all of the literals of the clause. This sorting method is by default enabled as it has positive influence on the speed with which MUSes are found.

The second automatic sorting method, *sort by length*, sorts the clauses with the shortest ones first and thereby focuses on removing subsumed clauses. A *subsumed clause* is a clause that is a superset of another clause in the formula. If the clauses are sorted using this method then the conditional addition of the shorter subsumed clauses will always precede the conditional addition of the subsuming clauses. The latter will never be added as the shorter clause implies the longer clause. Note that it is possible that a formula contains a MUS in which a clause C_i subsuming a clause C_j occurs as long as C_j does not, and that MUSes that have this property will not be found if clauses are sorted in this way. Sorting by length is also by default enabled and if both described clause sorting methods are used together then sorting by length has priority over sorting by weight.

To improve the speed with which the SAT problems are solved a heuristic was added which influences `MiniSat`'s branch direction heuristic. At each variable decision `MiniSat` branches to the negative side by default. In the first round of the algorithm this is left untouched. After the first round every clause that has not been removed has had an associated assignment in the previous round. As the clauses will remain in the same order in each round apart from the new critical clauses moving to the front this assignment can be seen as an estimate to the assignment we are looking for. Our program therefore sets the variable branch directions to those that would lead the solver in the direction of the associated assignment found in the previous round when it is looking for a new assignment.

4 Related Work

The argument of Lemma 2 is also used in work on finding subsets of infeasible linear programmes [16,17] and in recent work on the *Constraint Satisfaction Problem* (CSP), which is more generalised than SAT. In the latter work it is credited to [18]. The constraint that is critical due to Lemma 2 is called the *transition constraint* in the work on CSP, as it is on the transition from satisfiability to unsatisfiability.

Where we have focused on proving more clauses critical than only this transition constraint, or transition clause in our case, they focused on more efficient approaches to finding the transition constraint. Instead of adding a constraint to a sequence until it becomes unsatisfiable they do a binary search for the transition constraint.

The authors call their approach the *dichotomic approach*. It has the advantage of a logarithmic, rather than linear, worst case number of required SAT problems to solve per round. The authors applied this algorithm successfully to instances of CSP. Although its worst case required number of SAT problems to be solved is lower we reckon it has some disadvantages as an incremental SAT implementation is not immediate, at least not without adding clause selector variables that will make the problems considerably harder. Besides that only one critical clause will be found per round. Still, their approach is interesting and might perform well when implemented efficiently for SAT problems or even combined with some of the ideas presented here.

Another approach to finding an unsatisfiable core is applied by `zcore` [5]. It records which clauses are necessary to derive the empty clause in a resolution proof of unsatisfiability generated using a SAT solver (`zchaff` in this case). Those clauses form an unsatisfiable subset, but it might be far from minimal. To approximate a MUS closer the authors suggest to iterate `zcore` until the size of the unsatisfiable subset no longer reduces. We will refer to this approach by the name of the script supplied with `zchaff` for this purpose, which is `zruntillfix`. For finding a guaranteed MUS the authors of `zcore` supply `zminimal` which is an implementation of the naive MUS finding algorithm (1).

In a recent publication [7] an unsatisfiable core extractor called *Approximate One MUS* (`AOMUS`) was introduced. Our experimental results support the claim of the authors that it has good performance at the useful task of reducing benchmarks that describe FPGA routing problems. As the name indicates the "approximated MUSes" found by `AOMUS` are not guaranteed to be MUSes. The algorithm `OMUS`, which does guarantee the output of exactly one MUS, is the `AOMUS` algorithm with the addition of a post processor with the slightly understated name "fine tune". This procedure is in fact an implementation of the naive MUS finding algorithm (1).

The core extractor `AMUSE` [6] is best suited for use with formulas that contain unsatisfiable cores that are small compared to the formula size. It does not guarantee finding a MUS. `AMUSE`'s preference for small cores is supported by our experimental results and caused by the fact that it builds up the unsatisfiable subset by adding clauses to a satisfiable subset until it becomes unsatisfiable.

In [19] an approach for finding a MUS by using the resolution proof of the unsatisfiability of the input formula is presented. The algorithm presented there tests for every clause in the input formula whether the empty clause can still be derived from the original resolution proof after removing that clause and all clauses derived from it.

The *minimal unsatisfiability prover* (`MUP`) [20] is targeted at proving the input formula to be a MUS rather than at extracting MUSes from the input formula. It is meant as a post processor to core extractors that are able to give close approximation of MUSes, like `AMUSE`, `AOMUS` or `zruntillfix`.

5 Results

We have tested our implementation using a computing cluster with 20 nodes that each have 2 Intel Xeon 5130 (2Ghz) Dual Core processors, making up for a total of 80 processor cores. None of the tested MUS finders is based on a parallelised algorithm so each run was executed on a single processor core. All tests were run with an 1800 seconds time limit and a 2GB memory limit.

We used benchmarks describing various sorts of problems. We generated random 3-SAT formulas with 50 variables and 215 clauses, 100 variables and 430 clauses and 200 variables and 860 clauses. From each of the three sets we took 50 unsatisfiable formulas. We also added all unsatisfiable instances found in the DaimlerChrysler benchmark set which describes problems from automotive product configuration[1]. In the DaimlerChrysler benchmarks the unsatisfiable cores

[1] `http://www-sr.informatik.uni-tuebingen.de/~sinz/DC`

are only a small fraction of the formula size. Next we added the Bevan family from the handmade category of the SAT Competition[2] held in 2003. All benchmarks in the Bevan family are already MUSes. Finally, we put the benchmark set used in the paper presenting AOMUS together[3] we will refer to that set as *FPGA+Various*. Note that some FPGA routing problems occur both in original form and shuffled in that set [21].

Besides testing the MUS extraction qualities of MiniUnsat we have also tested it as a post processor to the output of AMUSE, AOMUS and zruntillfix. Those three programs were also tested using the naive MUS proving approach as a post processor. For AMUSE and zruntillfix a naive MUS prover implementation was found in zminimal. OMUS is the implementation of AOMUS followed by an internal naive MUS prover.

Although AOMUS outputs a core that is only an approximation to a MUS it may prove some clauses of that core critical. Those clauses are not tested again by the post processor implemented in OMUS. With permission of the author of AOMUS we have modified it to pass the information about those clauses to MiniUnsat when we used it with AOMUS.

We also tested MUP as a post processor to the three core extractors. We do not present the results here as MUP does not seem to be robust enough. The dtree generator supplied with MUP for generating the required *binary decision diagram* (BDD) often crashes. Fortunately, the c2d generator [22], which was suggested as an alternative by the author of MUP in a personal communication, works better. However, the number of successful runs using MUP with either of the two generators is much smaller than that of the other tested approaches, and where it is succesful it is not significantly faster either. The author of MUP uses a BDD variable reordering tool called MINCE [23] as a preprocessor for some of his benchmarks results, which might explain the difference between his and our results.

The results presented in Table 1 are meant to give a general impression of the performance of different approaches. Please note that only those benchmarks in which a MUS was found using all seven presented approaches were included in the calculations of the average MUS sizes. The run times and MUS sizes for each tested benchmark on all tested approaches, including MUP, are available on the internet[4]. Table 2 shows the average number of clauses that are proven critical in one round by making use of the associated assignment technique we described. From the results in this table one can easily see that checking if a clause is critical by testing if its successors are satisfied by its associated assignment leads to a significant reduction in the number of required rounds.

The six scatter plots that together form Fig 3 give an impression of the run times in seconds of the various approaches to extracting a MUS. Each scatter plot compares two program versions. In each scatter plot there is a data point for every benchmark, with the position along the horizontal axes indicating the run time of the approach labelled on the horizontal axis, and the vertical position

[2] http://www.satcompetition.org
[3] SAT Competitions and http://www.aloul.net/benchmarks.html
[4] http://www.tcs.hut.fi/~swiering/musfinding

Table 1. Results summary

`A+zmin`	`AMUSE + zminimal`
`A+M`	`AMUSE + MiniUnsat`
`z+zmin`	`zruntillfix + zminimal`
`z+M`	`zruntillfix + MiniUnsat`
`O`	`OMUS`
`AO+M`	`AOMUS + MiniUnsat`
`M`	`MiniUnsat`

Number of formulas a MUS was extracted from within 1800 seconds								
Set	#	`A+zmin`	`A+M`	`z+zmin`	`z+M`	`O`	`AO+M`	`M`
3-SAT 50 vars	50	50	50	50	50	50	50	50
3-SAT 100 vars	50	48	50	50	50	50	50	50
3-SAT 200 vars	50	6	50	8	50	48	50	50
DaimlerChrysler	84	84	84	84	84	84	84	84
Bevan	56	29	29	31	31	43	41	56
FPGA+Various	36	21	24	20	25	28	29	23
Sum	326	238	287	243	290	303	304	313

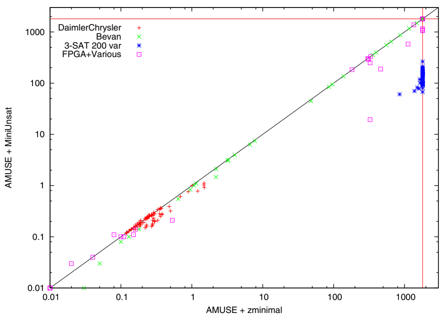
Average MUS size for formulas a MUS was extracted from by all approaches								
Set	#	`A+zmin`	`A+M`	`z+zmin`	`z+M`	`O`	`AO+M`	`M`
3-SAT 50 vars	50	101.4	96.2	95.2	94.8	92.4	92.3	101.3
3-SAT 100 v.	48	268	243.1	247.6	233.7	234	232.2	252.9
3-SAT 200 v.	3	624	563.3	639.3	576.3	546.3	546.7	578.3
DaimlerChr.	84	78.4	78.4	76.8	76.8	77.8	76	76.4
Bevan	28	186.4	186.4	186.4	186.4	186.4	186.4	186.4
FPGA+Various	15	225.5	220.4	231.9	217.3	226.4	220.6	221.5

Table 2. Average number of clauses proven critical per round

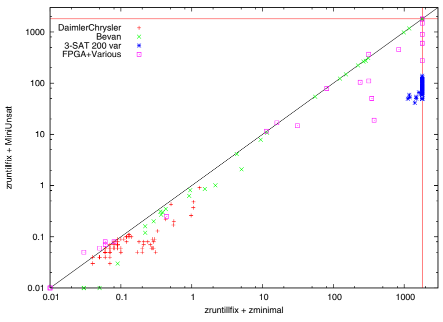
3-SAT 50 vars	2.6
3-SAT 100 vars	2.7
3-SAT 200 vars	2.8
DaimlerChrysler	10.2
Bevan	45.1
FPGA+Various	4.6

indicating the run time of the approach labelled on the vertical axis. Note that in all six plots both axes have a logarithmic scale. A value of 1800 seconds corresponds to a timeout.

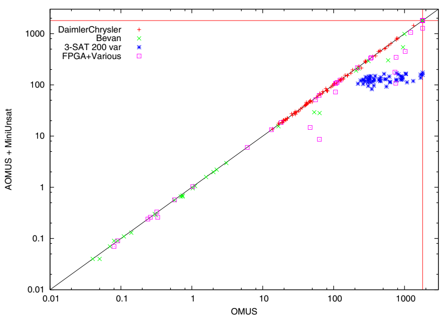
The first three scatter plots show the gains the core extractors `AMUSE` (a), `zruntillfix` (b) and `AOMUS` (c) have from using `MiniUnsat` rather than a naive MUS proving approach as a post processor. The improvement caused by using `MiniUnsat` when regarding the combination of core extractor and prover as one program is quite remarkable, especially for the random 3-SAT formulas with 200



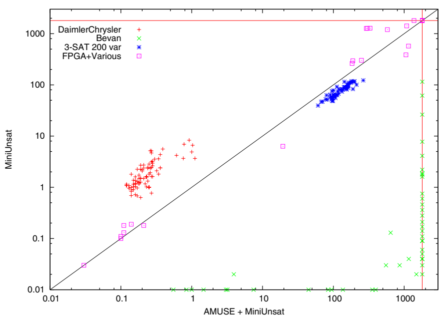
(a) `A+zmin` vs `A+M`



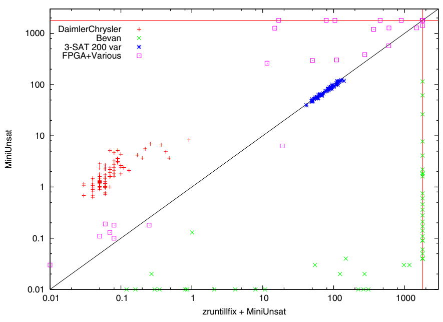
(b) `z+zmin` vs `z+M`



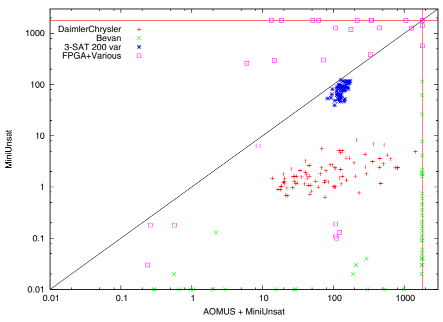
(c) `O` vs `AO+M`



(d) `A+M` vs `M`



(e) `z+M` vs `M`



(f) `AO+M` vs `M`

Fig. 3. Results

variables. While `AMUSE+zminimal` and `zruntillfix+zminimal` are not capable of extracting a MUS in half an hour from the majority of those formulas that goal can always reached when using `MiniUnsat` instead of `zminimal`.

The other scatter plots show how the three core extractors combined with `MiniUnsat` and regarded as one program perform against `MiniUnsat` without a preprocessor. From those scatter plots it can be seen that for the set of DaimlerChrysler benchmarks it pays off to use `AMUSE` (d) or `zruntillfix` (e) as a preprocessor. However, for the Bevan benchmarks using those preprocessors will mean most benchmarks will not be solved because the preprocessor times out. The performance of the `AOMUS` core extractor (f) as a preprocessor for the Bevan benchmarks is, unsurprisingly, similar to the other two tested preprocessors. However, in `AOMUS`'s scatter plot, on the horizontal line indicating a timeout for `MiniUnsat` we see a number of benchmarks from the set *FPGA+Various*. The benchmarks from that set that `MiniUnsat` fails to extract a MUS from without the help of `AOMUS` are FPGA routing problems, a domain in which `AOMUS` excels.

6 Conclusion

Although over the last years attention has been paid to the development of tools for extracting unsatisfiable subsets from unsatisfiable Boolean formulas most existing tools do not guarantee that the extracted unsatisfiable subsets are minimal. The `MiniUnsat` MUS finder we presented is capable of extracting a MUS from a wide range of unsatisfiable formulas at very competitive speeds.

In case the user wants to tune performance for a specific set of benchmarks several interesting combinations with existing software are recommendable. For example, if the program is applied in a setting were a MUS is often only a small fraction of the size of the formula it is extracted from, it is wise to use `AMUSE` as a preprocessor to `MiniUnsat`. For finding minimal unsatisfiable subsets in FPGA routing problems the use of `AOMUS` as a preprocessor is recommended.

Acknowledgements. Thanks to all developers of the tested tools for making their software available, either on the web or at request. Thanks to all colleagues that supported this work by showing their interest or commenting. Special thanks to Keijo Heljanko for all his help throughout this project.

References

1. Nam, G.J., Sakallah, K.A., Rutenbar, R.A.: Satisfiability-based layout revisited: Detailed routing of complex FPGAs vis search-based Boolean SAT. In: FPGA, pp. 167–175 (1999)
2. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) ETAPS 1999 and TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
3. McMillan, K.L.: Interpolants and symbolic model checking. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 89–90. Springer, Heidelberg (2007)
4. Rintanen, J., Heljanko, K., Niemelä, I.: Planning as satisfiability: parallel plans and algorithms for plan search. Artif. Intell. 170(12-13), 1031–1080 (2006)

5. Zhang, L., Malik, S.: Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formula. In: Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8 (2003)
6. Oh, Y., Mneimneh, M.N., Andraus, Z.S., Sakallah, K.A., Markov, I.L.: AMUSE: A minimally-unsatisfiable subformula extractor. In: Malik, S., Fix, L., Kahng, A.B. (eds.) DAC, pp. 518–523. ACM, New York (2004)
7. Grégoire, É., Mazure, B., Piette, C.: Local-search extraction of MUSes. Constraints 12(3), 325–344 (2007)
8. Lynce, I., Silva, J.P.M.: On computing minimum unsatisfiable cores. In: H. Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, Springer, Heidelberg (2005)
9. Liffiton, M.H., Sakallah, K.A.: On finding all minimally unsatisfiable subformulas [25], 173–186
10. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning 40(1), 1–33 (2008)
11. Wieringa, S.: Finding cores using a Brouwer's fixed point approximation algorithm. Master's thesis, Delft University of Technology, Faculty of EWI (2007)
12. van Maaren, H.: Pivoting algorithms based on Boolean vector labeling. Acta Mathematica Vietnamica 22(1), 183–198 (1997)
13. Kullmann, O., Lynce, I., Marques-Silva, J.: Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel. [24] 22–35
14. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
15. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. [15] 61–75
16. Tamiz, M., Mardle, S.J., Jones, D.F.: Detecting IIS in infeasible linear programmes using techniques from goal programming. Computers & OR 23(2), 113–119 (1996)
17. Galinier, P., Hertz, A.: Solution techniques for the large set covering problem. Discrete Applied Mathematics 155(3), 312–326 (2007)
18. de Siqueira, N.J.L., Puget, J.-F.: Explanation-based generalisation of failures. In: ECAI, pp. 339–344 (1988)
19. Dershowitz, N., Hanna, Z., Nadel, A.: A scalable algorithm for minimal unsatisfiable core extraction. [24] 36–41
20. Huang, J.: MUP: A minimal unsatisfiability prover. In: Tang, T.A. (ed.) ASP-DAC, pp. 432–437. ACM Press, New York (2005)
21. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Solving difficult instances of Boolean satisfiability in the presence of symmetry. IEEE Trans. on CAD of Integrated Circuits and Systems 22(9), 1117–1137 (2003)
22. Darwiche, A.: New advances in compiling CNF into decomposable negation normal form. In: de Mántaras, R.L., Saitta, L. (eds.) ECAI, pp. 328–332. IOS Press, Amsterdam (2004)
23. Aloul, F.A., Markov, I.L., Sakallah, K.A.: MINCE: A static global variable-ordering heuristic for SAT search and BDD manipulation. J. UCS 10(12), 1562–1596 (2004)
24. Biere, A., Gomes, C.P. (eds.): SAT 2006. LNCS, vol. 4121. Springer, Heidelberg (2006)
25. Bacchus, F., Walsh, T. (eds.): SAT 2005. LNCS, vol. 3569. Springer, Heidelberg (2005)