Accelerating MUS Extraction with Recursive Model Rotation

Anton Belov
Complex and Adaptive Systems Laboratory
University College Dublin
Email: anton.belov@ucd.ie

Joao Marques-Silva
Complex and Adaptive Systems Laboratory
University College Dublin
Email: jpms@ucd.ie

***Abstract*—Minimally Unsatisfiable Subformulas (MUSes) find a wide range of practical applications. A large number of MUS extraction algorithms have been proposed over the last decade, and most of these algorithms are based on iterative calls to a SAT solver. In this paper we introduce a powerful technique for acceleration of MUS extraction algorithms called *recursive model rotation* — a recursive version of the recently proposed model rotation technique. We demonstrate empirically that recursive model rotation leads to multiple orders of magnitude performance improvements on practical instances, and pushes the performance of our MUS extractor MUSer2 ahead of the currently available MUS extraction tools.**

I. Introduction

A minimally unsatisfiable subformula (MUS) of an unsatisfiable CNF formula F is any minimal, with respect to set inclusion, subset of clauses in F that is still unsatisfiable. MUSes find a wide range of practical applications. For example, MUS are used in a number of verification tasks to extract a concise description of inconsistency. As a result, development of effective MUS extraction algorithms is currently a very active area of research — examples of most recent work include [1], [2], [3], [4], [5], [6]. MUS algorithms can be roughly categorized as *constructive* (or, insertion-based), as *destructive* (or, deletion-based), or dichotomic. At the moment, destructive and hybrid MUS extraction algorithms outperform other approaches by a wide margin [1]. However, irrespective of the approach, the main bottleneck of MUS algorithms is the number of repeated calls to a SAT oracle.

Model rotation [1] is a method for detection of clauses that are included in all MUSes of a given formula via the analysis of models returned by a SAT oracle. While theoretically the technique does not guarantee the reduction in the number of SAT calls, in practice it does, and has been reported in [1] to provide for considerable performance gains (up to a factor of 5). In this paper we push the idea further — we propose a modification to the technique that in practice results in a very large reduction in the number of invocations of SAT oracle, significantly boosting the performance of our MUS extractor MUSer2. One of the key aspects of the proposed technique, which we call *recursive model rotation*, is that in principle it could be used with any type of MUS extraction algorithm.

Model rotation and recursive model rotation are described in Section III, following the necessary background in Section II. In Section IV we demonstrate the power of the new technique empirically. The paper is concluded in Section V with the discussion of some of the related work and possible enhancements to the proposed techniques.

II. Background

While we assume the familiarity with classical propositional logic (CPL) and SAT, here we clarify and fix the terminology used in this paper. We focus on formulas in CNF (*formulas*, from hence on), which we treat as (finite) sets of clauses. We assume that clauses do not contain duplicate variables.

Given a formula F we denote the set of variables that occur in F by $Var(F)$, and the set of variables that occur in $C \in F$ by $Var(C)$. An *assignment* h for F is a map $h : Var(F) \to \{0, 1\}$. By $h|_{\neg x}$ be denote the assignment $(h \setminus \{\langle x, h(x)\rangle\}) \cup \{\langle x, 1-h(x)\rangle\}$. Assignments are extended to formulas according to the semantics of CPL. If $h(F) = 1$, then h is a *model* of F. If a formula F has (resp. does not have) a model, then F is *satisfiable* (resp. *unsatisfiable*) and we write $F \in SAT$ (resp. $F \in UNSAT$).

By $Unsat(F, h)$ we denote the set of clauses falsified by h, i.e. $Unsat(F, h) = \{C \mid C \in F \text{ and } h(C) = 0\}$. If $F \in UNSAT$ and for some clause $C \in F$, we have $F \setminus \{C\} \in SAT$, then C is called a *transition clause* for F.

Formula F is called *minimally unsatisfiable*, in symbols $F \in MU$, if $F \in UNSAT$ and for any $F' \subset F$, $F' \in SAT$. Equivalently, F is minimally unsatisfiable, if every clause in F is a transition clause. Given a formula $F \in UNSAT$, a *minimally unsatisfiable subformula* of F, in symbols $MUS(F)$, is any $F' \subseteq F$ such that $F' \in MU$. In general, a given unsatisfiable formula may have more than one MUS. Clauses that belong to *all* MUSes of F are called *necessary* for F [7] — clearly, every transition clause is necessary and vice versa, as such the terms are often used interchangeably.

Most of the algorithms computing MUSes rely on the identification of the necessary clauses of the input formula F or its subformulas. In the constructive approach the clauses of F are added to an initially empty set F' until F' becomes unsatisfiable. Then, the last clause added to F' is necessary for F'. In the destructive approach the clauses are removed from F until the resulting formula F' becomes satisfiable. Then, the last clause C removed from F is necessary for $F' \cup \{C\}$. We refer the reader to [1] for an overview of the state-of-the-art in MUS computation algorithms.

III. Recursive Model Rotation

Let F be an unsatisfiable formula, let $C \in F$ be a transition clause, and let h be a model of $F \setminus \{C\}$. Note that this implies $Unsat(F, h) = \{C\}$. In fact, we have the following observation [1]:

Proposition 1: Let F be an unsatisfiable formula. Then $C \in F$ is a transition clause if and only if there exists an assignment h such that $Unsat(F, h) = \{C\}$.

Proof: C is a transition clause iff $F \setminus \{C\} \in SAT$ iff there exists an assignment h such that $Unsat(F \setminus \{C\}, h) = \emptyset$ iff $Unsat(F, h) = \{C\}$. ■

Model rotation exploits Proposition 1 to detect additional transition clauses during the computation of MUS. Assuming the destructive approach, let C be a transition clause detected by invoking a SAT solver on the formula $F' = F \setminus \{C\}$ — that is, the SAT solver determined that F' is satisfiable, and returned a model h of F', while F is known to be unsatisfiable. A similar situation occurs in the constructive approach when C is the clause that caused unsatisfiability of the current subset of F. In most MUS extraction algorithms the model h is discarded (a notable exception is MiniUnsat [5] — in Section V we discuss the relationship of our technique with this algorithm). However, consider the assignment $h' = h|_{\neg x}$ for some $x \in Var(C)$. Clearly, $C \notin Unsat(F, h')$, but $Unsat(F, h') \neq \emptyset$. Furthermore, if the set $Unsat(F, h')$ contains *exactly one* clause C', then by Proposition 1, C' is a transition clause. The model h' (of $F \setminus \{C'\}$) is said to be obtained by the *rotation* of the model h with respect to clause C and variable x. Note that now, the model h' can be rotated again — this time, with respect to clause C' and some variable $x' \in C'$ — possibly giving another transition clause. Note that x' should be different from x, as otherwise the rotation will give the initial model h.

Example 1: Let $F = \{C_1, \ldots, C_5\}$, where

$$C_1 = \neg x_1 \vee \neg x_2 \qquad C_3 = x_2 \vee \neg x_3 \qquad C_5 = x_1 \vee x_2$$
$$C_2 = x_1 \vee \neg x_2 \qquad C_4 = x_2 \vee x_3$$

The formula F is unsatisfiable, and the clause C_1 is a transition clause for F, i.e. $F_1 = F \setminus \{C_1\} \in SAT$. Let $h_1 = \{x_1, x_2, x_3\}$ be the model of F_1 returned by a SAT solver. We have $Unsat(F, h_1) = \{C_1\}$. Let $h_2 = h_1|_{\neg x_1}$, that is $h_2 = \{\neg x_1, x_2, x_3\}$. We have $Unsat(F, h_2) = \{C_2\}$, and therefore, C_2 is another transition clause.

In model rotation the process is continued until for some clause C_f and the corresponding model h_f of $F \setminus \{C_f\}$, for every variable $x \in C_f$ the set $Unsat(F, h|_{\neg x})$ is either not a singleton, or contains a clause that is already known to be a transition clause. In the above example, the rotation stops at h_2 as $Unsat(F, h_2|_{\neg x_2}) = \{C_3, C_5\}$.

This stopping criterion guarantees that the total number of model rotations during the execution of an MUS computation algorithm on formula F is at most $|M| \cdot c_{max}$, where M is the computed MUS of F, and c_{max} is the maximum among the lengths of clauses in M. On the other hand, each successful model rotation (i.e. the one that detects a new transition clause) saves a potentially expensive call to a SAT solver. Given that in practical instances the size of MUSes rarely exceeds a few tens of thousands of clauses, it is not surprising that model rotation often provides for significant performance gains — in Section IV we demonstrate these gains empirically.

We now note that in model rotation *at most one variable* from each necessary clause C is used for rotation — the original motivation in [1] was to keep the model rotation a low overhead technique. We observe, however, that the stopping criterion described above still guarantees that number of rotations is linear in the size of the computed MUS, even if *all variables* from C are used for rotation. On the other hand, we have the following result:

Proposition 2: Let F be an unsatisfiable formula, let $C \in F$ be a transition clause, and let h be a model of $F \setminus \{C\}$. Then, the sets $Unsat(F, h|_{\neg x})$ for $x \in Var(C)$ are pairwise disjoint.

Proof: Let x be a variable in C, and let C' be some clause in $Unsat(F, h|_{\neg x})$. Since $C' \notin Unsat(F, h)$, the literal of variable x was *critical* in C' under h (that is, the only literal in C' that evaluates to 1 under h). Since every clause has *at most* one critical literal, the fact follows. ■

Hence, by performing model rotation on different variables of C we are *guaranteed* to obtain disjoint sets of clauses, thus increasing the likelihood of detecting additional transition clauses. Consider again the formula F in Example 1, and assume that the model rotation stopped at assignment $h_2 = \{\neg x_1, x_2, x_3\}$ due to the stopping criterion. We can now "backtrack" to the assignment h_1 and attempt to rotate h_1 with respect to C_1 on variable x_2. The rotation results in the assignment $h_2' = \{x_1, \neg x_2, x_3\}$, and since $Unsat(F, h_2') = \{C_3\}$ we have a new transition clause C_3. Rotation of h_2' on x_3 results in the assignment $h_3 = \{x_1, \neg x_2, \neg x_3\}$, which gives another transition clause C_4. Rotating h_3 on x_2 results in the assignment $h_4 = \{x_1, x_2, \neg x_3\}$ at which point the rotation terminates, because $Unsat(F, h_4) = \{C_1\}$ and C_1 is already known to be a transition clause. In this example, such *recursive* model rotation allows to detect all of the transition clauses of F. Remarkably, as demonstrated in Section IV, the cases when the recursive model rotation finds all, or close to all, of the necessary clauses do occur often on practical instances. The sketch of the algorithm for the recursive model rotation is presented in Algorithm 1. The algorithm is invoked whenever an MUS extractor detects a new transition clause as a result of a call to a SAT solver.

We now note that the "if" direction of Proposition 1 can be generalized as follows:

Proposition 3: Let F be an unsatisfiable formula. Then, for any assignment h the set $Unsat(F, h)$ contains at least one clause from each of the MUSes of F.

Proof: If not, then the set $F \setminus Unsat(F, h)$ includes an MUS of F, and so must be unsatisfiable. ■

Proposition 3 justifies the following heuristic for selection of clauses in deletion-based MUS extraction algorithms: whenever the recursive model rotation arrives at an assignment h with $|Unsat(F, h)| > 1$, try to remove the clauses from $Unsat(F, h)$ next. The idea is that for instances with many

Algorithm 1 RECURSIVE-MODEL-ROTATION(F, M, C, h)

Input: F — an unsatisfiable CNF formula
$M \subseteq F$ — a set of transition clauses of F
$C \in M$ — a transition clause
h — a model of $F \setminus \{C\}$
Effect: M may contain additional transition clauses of F

1: **for all** $x \in Var(C)$ **do**
2: $h' \leftarrow h|_{\neg x}$
3: **if** $Unsat(F, h') = \{C'\}$ **and** $C' \notin M$ **then**
4: $M \leftarrow M \cup \{C'\}$
5: RECURSIVE-MODEL-ROTATION(F, M, C', h')
6: **end if**
7: **end for**

MUSes chances are that the clauses from this set belong to different MUSes, and so among the next few calls to the SAT solver, there will be UNSAT results — these, in turn, are beneficial for the deletion algorithms which use clause set refinement [1] to remove the clauses outside of unsatisfiable core. Our experiments, reported in Section IV, confirm that this rotation-based re-ordering of clauses is indeed beneficial on some classes of problems.

IV. EXPERIMENTAL STUDY

The algorithm for recursive model rotation (RMR) is implemented in our new MUS extractor MUSer2. MUSer2 is a slightly optimized version of MUSer — an MUS extractor from [1]. Both MUSer2 and MUSer implement HYB — a deletion-like MUS computation algorithm that employs clause set refinement, redundancy removal and (non-recursive) model rotation. The results of the experimental evaluation of HYB reported in [1] show clearly that it significantly outperforms all of the publicly available MUS extraction algorithms — we also reproduce some of the data from [1] in this section.

To evaluate the effectiveness of recursive model rotation we performed a comprehensive experimental study on 500 benchmarks submitted to the MUS track of SAT Competition 2011 (http://www.satcompetition.org/2011) — this is also the same set of instances that was used in [1] to evaluate HYB. The benchmark instances were obtained from various industrial applications of SAT, including hardware bounded model checking, FPGA routing, hardware and software verification, equivalence checking, abstraction refinement, design debugging, functional decomposition and bioinformatics. Note that the benchmarks are pre-processed via trimming [1]. The benchmarks are available for download at http://logos.ucd.ie/~jpms/Drops/SAT11. The experiments were performed on an HPC cluster, where each node is dual quad-core Xeon E5450 3 GHz with 32 GB of memory. The timeout was set at 1200 seconds, and memory was limited at 4 GB.

Figure 1 presents the incremental-time plot that provides an overview of the results of our experimental study. The plot contains data from [1] for SAT4J [8] MUS extractor in destructive mode (SAT4J-D), a destructive MUS algorithm Picomus from the Picosat distribution [9], and the algorithm

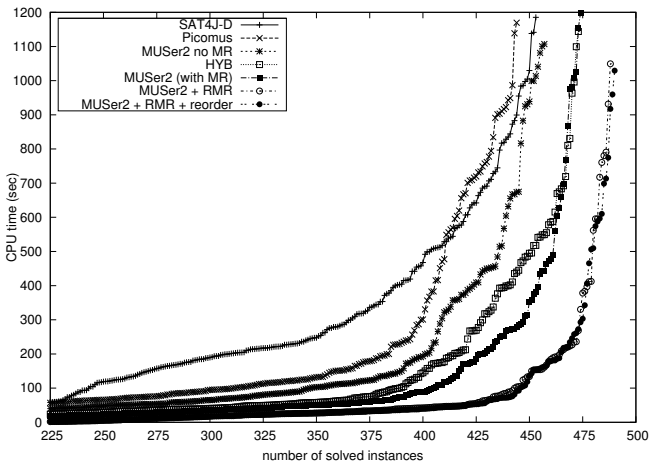

Fig. 1. Incremental-time plot showing data for all MUS extractors. To keep the plot legible, the data for the first 225 instances in not shown.

TABLE I
SUMMARY OF THE PERFORMANCE OF THE VARIANTS OF MUSER2

MUSer2 variant	Solved (from 500)	Total time (sec)	Total SAT calls	Avg. MUS size (% of input size)	Avg. % of tr. clauses by rot
no MR	457	44227	1741954	93.44	n/a
with MR	474	20249	797490	93.45	61.33
+RMR	488	12609	570303	93.46	77.57
+RMR+reorder	**490**	**12153**	570867	93.44	77.44

Columns 3-6 contain data for instances solved by *all* variants of MUSer2. The last column shows the percentage of transition clauses in the computed MUS that were detected by model rotation.

HYB implemented in MUSer. The former two are the top performing publicly available MUS extractors among those evaluated in [1]. In addition, the plot contains data for: MUSer2 with model rotation disabled (MUSer2 no MR), MUSer2 (with model rotation), MUSer2 with recursive model rotation (MUSer2 + RMR), and, finally, MUSer2 + RMR with rotation-based reordering of clauses described in Section III.

The following conclusions can be drawn. First, we note that MUSer2 has a clear performance edge on the currently publicly available MUS extraction tools. Second, the comparison between MUSer2 and MUSer2 without model rotation demonstrates that model rotation, even in the form introduced in [1], is an extremely powerful acceleration technique for MUS extraction algorithms. Third, we note that recursive model rotation increases the power of model rotation further. Finally, it appears that rotation-based clause re-ordering heuristic might also result in a slight performance edge.

Table I provides additional evidence of the power of model rotation in general, and recursive model rotation in particular. The table presents the number of solved instances for each of the variants of MUSer2 in the second column. The data in the remaining columns is for instances solved by all of the four variants. We note that the addition of recursive model rotation results in significant reduction in overall MUS extraction time, and allows to solve significantly more instances. We also point out that RMR is really a clear-win technique for MUS extraction algorithms as the size of computed MUSes is not affected negatively. Finally, the last column in Table I shows

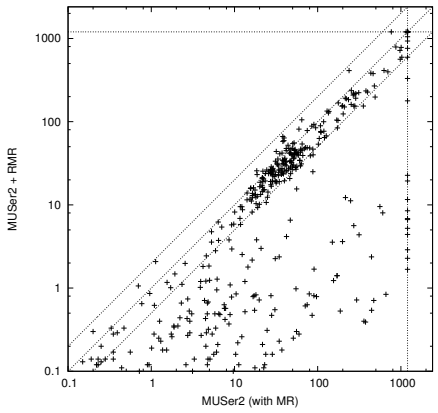

Fig. 2. MUSer2 with model rotation vs. recursive model rotation: CPU time.

the percentage of MUS clauses detected by model rotation — over 77% on average for RMR.

Both Figure 1 and Table I show that the addition of recursion to model rotation results in a significant performance edge. Nevertheless, we present additional scatter plots comparing the two versions of the technique. The plot in Figure 2 shows that on many problems the addition of recursion results in 1, 2 and in some cases 3 orders of magnitude speed-ups. The analysis of our experimental data suggests that recursion allows to detect *significantly* more necessary clauses, in many cases all of them.

The effect of recursive model rotation on some of the specific classes of instances from our benchmark set are demonstrated in Table II — *debug* refers to design debugging instances, *decomp* refers to functional decomposition instances (cf [10]), *ibm* refers to benchmarks from IBM, and *intel* are the abstraction-refinement benchmarks from [6]. While for some of the classes the performance improvements are moderate (although exceeding a factor of 2), on instances from design debugging and abstraction refinement we observe over 2 orders of magnitude speed-ups.

Finally, in Table III we compare the performance of MUSer2 with that of the resolution-based destructive MUS extraction algorithm 1MN [6]. Since the solver that implements the latter is not publicly available, we reproduce the data from [6]. For this experiment we used the original, *untrimmed* instances, and set the timeout to 2 hours, as in [6], and memout to 4 GB. The hardware used in our experiments appears to be similar to the one reported in [6]. We note that with the exception of two 4pipe instances, MUSer2 has a clear performance advantage over 1MN — on some instance (longmult) we observe 2 orders of magnitude speed-ups. On 4pipe MUSer2 is slower by a factor of 1.5, and on 4pipe_k it runs out of memory. The reason for the latter is the SAT solver used in this set of experiments (picosat-935), as our techniques have negligible memory overhead. It should be noted that MUSer2 uses SAT solver as a black-box (as opposed to resolution-based approach of [6] which requires significant modifications to a SAT solver). We suspect that switching to a more efficient SAT solver will resolve this issue.

TABLE II
MUSER2: NO ROTATION VS. RECURSIVE ROTATION, SELECTED CLASSES

Class (total num)	no rotation			recursive rotation			
	num solv.	time (sec)	SAT calls	num solv.	time (sec)	SAT calls	% by rot.
debug(120)	103	7041	129651	**120**	**65**	5713	94.82
fdec(143)	143	9738	874946	143	**4679**	307422	64.79
ibm(45)	42	5156	204134	42	**2255**	66563	84.38
intel(49)	41	7441	31640	**48**	**38**	346	97.27

TABLE III
MUSER2 + RMR VS. 1MN [6]: CPU TIME (SEC)

Instance	1MN	MUSer-2 + RMR	MUSer-2 + RMR + reorder
4pipe	**1417**	2101	1776
4pipe_1_ooo	1528	**425**	477
4pipe_2_ooo	2383	**1070**	1227
4pipe_3_ooo	2560	**593**	779
4pipe_4_ooo	2432	**568**	600
3pipe_k	167	104	**90**
4pipe_k	**1426**	MO	MO
barrel5	68	**9**	10
barrel6	348	**95**	150
barrel7	849	115	**103**
barrel8	4115	**1270**	2338
longmult4	14	**0.4**	**0.4**
longmult5	143	2.5	**1.6**
longmult6	968	13	**10**
longmult7	5099	103	**40**

V. CONCLUDING REMARKS

The analysis of models returned by a SAT solver during MUS extraction can be attributed to [5], where it was explored in the context of constructive MUS extractor MiniUnsat. Nevertheless, model rotation differs fundamentally from this work. In our view, recursive model rotation (and clause reordering) is just one example of structure-based techniques for MUS extraction. – we are currently investigating additional techniques of similar nature.

Acknowledgements: We thank the anonymous referees for helpful comments. This work is partially supported by SFI PI grant BEACON (09/IN.1/I2618).

REFERENCES

[1] J. Marques-Silva and I. Lynce, "On improving MUS extraction algorithms," in *Theory and Applications of Satisfiability Testing*, 2011.
[2] C. Desrosiers, P. Galinier, A. Hertz, and S. Paroz, "Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems," *J. of Combinatorial Optimization*, vol. 18, no. 2, 2009.
[3] J. Marques-Silva, "Minimal unsatisfiability: Models, algorithms and applications," in *Int'l Symposium on Multiple-Valued Logic*, 2010.
[4] É. Grégoire, B. Mazure, and C. Piette, "On approaches to explaining infeasibility of sets of Boolean clauses," in *Int'l Conference on Tools with Artificial Intelligence*, 2008.
[5] H. van Maaren and S. Wieringa, "Finding guaranteed MUSes fast," in *Theory and Applications of Satisfiability Testing*, 2008.
[6] A. Nadel, "Boosting minimal unsatisfiable core extraction," in *Formal Methods in Computer-Aided Design*, 2010.
[7] O. Kullmann, I. Lynce, and J. Marques-Silva, "Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel," in *Int'l Conference on Theory and Applications of Satisfiability Testing*, 2006.
[8] D. Le Berre and A. Parrain, "The Sat4j library, release 2.2," *J. on Satisfiability, Boolean Modeling and Computation*, vol. 7, 2010.
[9] A. Biere, "PicoSAT essentials," *J. on Satisfiability, Boolean Modeling and Computation*, vol. 2, 2008.
[10] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung, "Bi-decomposing large Boolean functions via interpolation and satisfiability solving," in *Design Automation Conference*, 2008.