Journal on Satisfiability, Boolean Modeling and Computation 8 (2012) 123-128

MUSer2: An Efficient MUS Extractor

SYSTEM DESCRIPTION

Anton Belov anton.belov@ucd.ie
CASL/CSI, University College Dublin
Dublin, Ireland
Joao Marques-Silva jpms@ucd.ie
CASL/CSI, University College Dublin, Dublin, Ireland
IST/INESC-ID, Lisbon, Portugal

Abstract

Algorithms for extraction of Minimally Unsatisfiable Subformulas (MUSes) of CNF formulas find a wide range of practical applications, including product configuration, knowledge-based validation, hardware and software design and verification. This paper describes the MUS extractor `MUSer2`. `MUSer2` implements a wide range of MUS extraction algorithms, integrates a number of key optimization techniques, and represents the current state-of-the-art in MUS extraction.

KEYWORDS: *Minimal unsatisfiability, MUS extraction, SAT applications*

Submitted June 2012; revised August 2012; published December 2012

1. Introduction

A minimally unsatisfiable subformula (MUS) of an unsatisfiable CNF formula $\mathcal{F}$ is any minimal, with respect to set inclusion, subset of clauses in $\mathcal{F}$ that is still unsatisfiable. MUSes find a wide range of practical applications. Some of the applications from the early 2000's that motivated the interest in algorithms for computing MUSes include type debugging in programming languages, circuit error diagnosis, and error localization in automotive product configuration data. However, by the late 2000's it became clear that some of the technologies that traditionally relied on the computation of non-minimal unsatisfiable subformulas of propositional formulas (the *unsatisfiable cores*) can benefit significantly, and are willing to pay the price, for the computation of MUSes. As a result, development of effective MUS extraction algorithms is currently a very active area of research.

This paper describes an MUS extractor `MUSer2`. `MUSer2` implements a number of MUS (and group-MUS) extraction algorithms and includes various important optimization techniques, such as clause-set refinement and model rotation. The tool is SAT solver agnostic — that is, it treats SAT solvers in a black-box manner — this allows for experimentation with various SAT solvers, and allows to easily capitalize on the advancements in SAT solving. The tool includes a built-in tester that allows to check whether a computed clause-set is indeed an MUS — this feature simplifies the implementation of new algorithms. As demonstrated in Section 3, `MUSer2` outperforms by a wide margin the top MUS extractors

This work is partially supported by SFI grant BEACON (09/IN.1/I2618), and by FCT grants ATTEST (CMU-PT/ELE/0009/2009) and POLARIS (PTDC/EIA-CCO/123051/2010).

from SAT Competition 2011. `MUSer2` is an open-source (GPLv3) software. The tool is available for download from http://logos.ucd.ie/wiki/doku.php?id=muser. A longer version of this paper, presented at Pragmatics of SAT 2012 workshop, is also available from the website, and includes a complete bibliography and additional experimental data.

We assume familiarity with propositional logic, its clausal fragment, and commonly used terminology and notation of the area of SAT. A CNF formula $\mathcal{F}$ is *minimally unsatisfiable* if (i) $\mathcal{F}$ is unsatisfiable, and (ii) for any clause $C \in \mathcal{F}$, the formula $\mathcal{F} \setminus \{C\}$ is satisfiable. We denote the set of minimally unsatisfiable CNF formulas by MU. A CNF formula $\mathcal{F}'$ is a *minimally unsatisfiable subformula (MUS)* of a formula $\mathcal{F}$ if $\mathcal{F}' \subseteq \mathcal{F}$ and $\mathcal{F}' \in \mathsf{MU}$. The set of MUSes of a CNF formula $\mathcal{F}$ is denoted by $\mathsf{MUS}(\mathcal{F})$. (In general, a given unsatisfiable formula $\mathcal{F}$ may have more than one MUS.) A clause $C \in \mathcal{F}$ is *necessary* for $\mathcal{F}$ if $\mathcal{F}$ is unsatisfiable and $\mathcal{F} \setminus \{C\}$ is satisfiable. Necessary clauses are often referred to as *transition* clauses. The set of all necessary clauses of $\mathcal{F}$ is precisely $\bigcap \mathsf{MUS}(\mathcal{F})$. Thus, $\mathcal{F} \in \mathsf{MU}$ if and only if every clause of $\mathcal{F}$ is necessary. The problem of deciding whether a given CNF formula is in MU is DP-complete [13]. Motivated by several applications, minimal unsatisfiability and related concepts have been extended to CNF formulas where clauses are partitioned into disjoint sets called *groups* [9, 12]. Throughout the paper, m denotes the number of clauses in the input CNF formula $\mathcal{F}$, $m = |\mathcal{F}|$, and k denotes the number of clauses in the largest MUS $\mathcal{M}$, $k = |\mathcal{M}|$. Over the years, three main approaches have been proposed for computing an MUS: *constructive* or *insertion-based* [5], *destructive* or *deletion-based* [4, 1] and *dichotomic* [7, 6]. Constructive approaches require $\mathcal{O}(m \times k)$ calls to an NP-oracle (e.g. a SAT solver), destructive approaches require $\mathcal{O}(m)$ calls, and dichotomic approaches require $\mathcal{O}(k \times \log m)$ calls. See [10] for a recent overview of MUS extraction algorithms.

2. MUSer2 — Functionality and Features

Prior to the overview of the functionality and the features of `MUSer2` we note that the core of the tool does not make any distinction between the CNF and the group-CNF formulas. All algorithms and optimization techniques that we describe in this section in terms of CNF formulas (resp. MUSes), are also implemented in `MUSer2` for group-CNF formulas (resp. group-oriented MUSes).

2.1 Algorithms

The current version of `MUSer2` implements the following MUS extraction algorithms.

The *hybrid* algorithm [11]. This is an instantiation of the deletion-based approach to MUS extraction: the necessary clauses are detected on transition from UNSAT to SAT, as in the deletion-based approach, however the MUS is built bottom-up, as in the insertion-based approach. The pseudocode of the algorithm is presented in [11]. The algorithm requires $O(m)$ SAT solver calls in the worst case, however in the presence of various optimizations described below, many practical instances are solved with significantly smaller number of calls, often as few as two.

The *dichotomic* algorithm [7, 6]. The worst-case number of SAT solver calls required by this algorithm is $\mathcal{O}(k \times \log m)$ (see details in [10]). Given the logarithmic dependency on the size of the input formula, and the linear dependency on the size of the *MUS*, the

algorithm might be expected to perform comparably or better than the hybrid approach in some cases. However, on practical instances this does not seem to be the case, with the dichotomic approach performing notably worse than the hybrid — see Fig. 1.

The *insertion-based*, or *constructive*, algorithm [5]. The detailed pseudocode of the algorithm is available in [10]. This algorithm requires $\mathcal{O}(m \times k)$ calls to a SAT solver in the worst case, and so, even in the presence of all relevant optimizations, does not scale on practical MUS extraction problems (see Fig. 1).

2.2 Optimization techniques

`MUSer2` includes the implementation of a number of optimization techniques, some of which, namely clause-set refinement and recursive model rotation, are *essential* for effective MUS extraction. The optimizations described below are applicable to all MUS extraction algorithms from Section 2.1.

Clause-set refinement and trimming. When, during the computation of an MUS, a SAT solver determines that a current subformula is unsatisfiable, the unsatisfiable core returned by the solver must contain at least one MUS. Thus, all clauses that fall outside of the core can be discarded. This optimization technique, referred to as *clause-set refinement* [11], is crucial for reducing the number of SAT solver calls in MUS extraction algorithms. Clause-set refinement can also be used as a *preprocessing* technique, i.e. prior to testing any of the clauses for the necessity. In this case, a SAT solver is invoked iteratively on computed unsatisfiable cores until no changes are detected between calls (cf. [16]). For large problem instances, iterating the computation until a fixed point can be inefficient, and so `MUSer2` implements a simpler alternative of iterating the computation for a constant number of times, or until the size change in the computed unsatisfiable subformulas is below a given threshold. We refer to this preprocessing technique a *clause-set trimming*.

Recursive model rotation (RMR). RMR [2] is an improved variant of *model rotation* — a powerful optimization technique introduced in [11]. Model rotation is initialized with an assignment to the variables of the current approximation of an MUS under which there is a single falsified clause. Such an assignment is a *witness* of the necessity of the clause. For deletion-based algorithms the witnesses are returned by the SAT solver whenever the removal of some clause makes the current working formula satisfiable. For insertion-based and dichotomic algorithms the witness is available on the termination of the inner loop. Given a witness, model rotation makes local changes to the truth-values in the witness with the attempt to obtain a witness for another clause, thus proving the clause necessary without a SAT solver call. In RMR this process is performed recursively — see [2] for details, the pseudocode, and an evaluation of the effectiveness of the technique.

Redundancy removal. This technique consists of adding extra constraints to the working formula prior to SAT solver calls. In the context of hybrid MUS extraction algorithm this is done by adding to the formula the negation of the removed clause (or the CNF representation of the negation of the removed group of clauses) prior to the call. This technique was first used in [15], in the context of a constructive MUS extraction algorithm, and in [11] in the context of the hybrid algorithm. Note the integration of redundancy removal and clause-set

refinement is not immediate, since the clauses from the redundancy removal technique can be part of the computed unsatisfiable core. We refer the reader to [11] for details.

2.3 SAT solver interface

One of the notable features of `MUSer2` is the fact that SAT solvers are treated in a black-box manner. That is, a SAT solver is accessed through a thin wrapper around the solver's API. The requirements from the API are minimal: SAT solvers that support incremental SAT are expected to be able to solve with assumptions, to return models, and in the case of unsatisfiable outcomes, to return the set of failed assumptions (this is needed for clause-set refinement and trimming). SAT solvers that do not support incremental SAT solving are expected to return models, and to provide an unsatisfiable core — the latter, again, only in the case refinement or trimming are used. The advantage of this design of `MUSer2` is that it allows to easily plug in and experiment with different SAT solvers, and even use different SAT solvers during the MUS extraction. Additional potential benefit of this design is that `MUSer2` may provide a platform for evaluation of the performance of incremental SAT solvers. The design, however, is not without a cost. For once, there is an extra layer of data-structures — `MUSer2` maintains its own copy of clauses of the formula. Additionally, `MUSer2` is unable to squeeze extra performance by making modifications inside SAT solver. However, we work on the premises that (*i*) most of the time in MUS extraction is spent in SAT solving, and (*ii*) it is very difficult to keep any particular SAT solver up-to-date on the newly developed SAT solving techniques, and so simply plugging in a faster SAT solver into the MUS extractor can be a more effective alternative.

The currently publicly available version of `MUSer2` includes the wrappers for SAT solvers `minisat-2.2.0` (`http://minisat.se/`) and `picosat-935` [3]. We are planning to add wrappers to additional SAT solvers in the near future.

2.4 Other features

Additional useful features of `MUSer2` include: (*i*) control over the ordering of clauses during MUS extraction; (*ii*) testing of the computed MUSes by means of a thoroughly externally tested implementation of the hybrid algorithm with refinement and RMR — the use of hybrid algorithm for MUS testing is motivated by the frequent requirement to test large MUSes within reasonable time limits; (*iii*) output of the computed MUSes (resp. group-MUSes) in CNF (resp. GCNF) formats; (*iv*) comprehensive statistics.

3. Experimental Data

In this section we present the results of experimental evaluation of `MUSer2` on the 295 benchmark instances from the MUS track SAT Competition 2011 [1]. The experiments were performed on an HPC cluster, where each node is dual quad-core Intel Xeon E5450 3 GHz with 32 GB of memory. Each algorithm was run with a timeout of 1800 seconds and a memory limit of 4 GB per input instance.

The cactus plot in Fig. 1 presents (*i*) the comparison of the performance of `MUSer2` with the top three MUS extractors from SAT Competition 2011, namely `MoUsSaka` [8] which

1. `http://www.satcompetition.org/`.

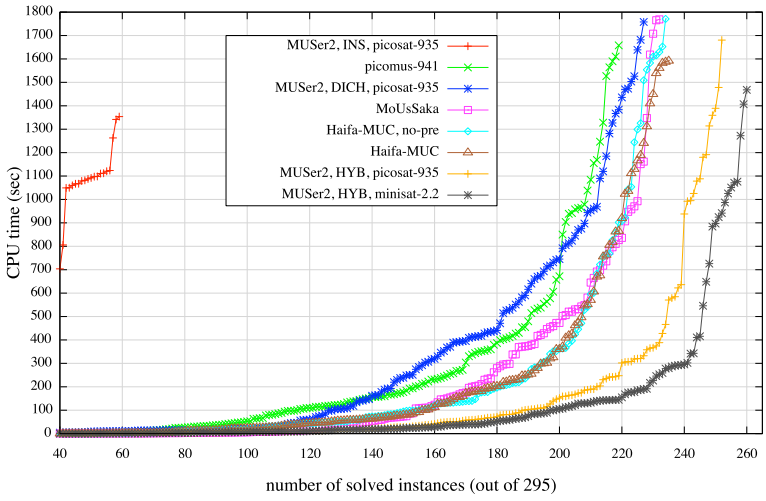

Figure 1. Runtimes of various extractors on the benchmarks from the MUS track of SAT Competition 2011. Time limit 1800 sec, memory limit 4 GB.

was ranked 3-rd, and `Haifa-MUC` [14] without (resp. with) preprocessing, which was ranked 1-st (resp. 2-nd); (*ii*) the comparison of the performance of `MUSer2` with `picomus-941` [3]; (*iii*) the comparison of the performance of hybrid MUS extraction algorithm with the dichotomic and the insertion-based MUS extraction algorithms in `MUSer2`; and, finally, (*iv*) the comparison of the performance of the hybrid algorithm in `MUSer2` using `minisat-2.2.0` and `picosat-935` [3] SAT solvers. In the plot, HYB refers to the hybrid algorithm, DICH to the dichotomic, and INS to the insertion-based algorithm. Clause-set trimming was not used in the experiments, and the redundancy removal technique is not yet implemented in the insertion-based and the dichotomic algorithms.

A number of observations can be made. First, we note that `MUSer2` significantly outperforms the winners of SAT Competition 2011. Beside solving extra 25 instances compared to `Haifa-MUC`, its notably faster than the three extractors on the problems of medium to high difficulty. Similarly, the comparison of `MUSer2` using `picosat-935` with `picomus-941` demonstrates clearly the effectiveness of the algorithms and optimizations implemented in `MUSer2`. We also conclude that the hybrid algorithm is significantly more effective that the insertion-based and the dichotomic algorithms, despite the optimizations integrated into these algorithms [2]. Finally, we note that in the context of `MUSer2` and the selected benchmark instances, `minisat-2.2.0` is a more effective SAT oracle than `picosat-935`.

4. Conclusions and Future Work

In this paper we presented a state of the art MUS extractor `MUSer2`. Future work on the tool includes implementation of additional MUS extraction algorithms and optimization techniques, and integration of additional SAT solvers.

2. The redundancy removal optimization is not implemented in the two algorithms, however, based on our experience, it is highly unlikely to close such a significant performance gap.

References

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

[2] A. Belov and J. Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *FMCAD*, 2011.

[3] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**:75–97, 2008.

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

[5] J. L. de Siqueira and J.-F. Puget. Explanation-based generalisation of failures. In *ECAI*, pages 339–344, 1988.

[6] F. Hemery, C. Lecoutre, L. Sais, and F. Boussemart. Extracting MUCs from constraint networks. In *ECAI*, pages 113–117, 2006.

[7] U. Junker. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *AAAI Conference on Artificial Intelligence*, pages 167–172, 2004.

[8] S. Kottler. Description of the SApperloT, SArTagnan and MoUsSaka solvers for the SAT-Competition 2011. http://www.satcompetition.org/2011/, 2011.

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

[10] J. Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications. In *ISMVL*, pages 9–14, 2010.

[11] J. Marques-Silva and I. Lynce. On improving MUS extraction algorithms. In *SAT*, pages 159–173, 2011.

[12] A. Nadel. Boosting minimal unsatisfiable core extraction. In *FMCAD*, October 2010.

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

[14] V. Ryvchin and O. Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *SAT*, pages 174–187, 2011.

[15] H. van Maaren and S. Wieringa. Finding guaranteed MUSes fast. In *SAT*, pages 291–304, 2008.

[16] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 10880–10885, 2003.