Proceedings of SAT Competition 2023 : Solver, Benchmark and Proof Checker Descriptions

2023

Balyo , T , Heule , M , Iser , M , Järvisalo , M & Suda , M (eds) 2023 , Proceedings of SAT Competition 2023 : Solver, Benchmark and Proof Checker Descriptions . Department of Computer Science Series of Publications B , vol. B-2023-1 , Department of Computer Science, University of Helsinki , Helsinki .

http://hdl.handle.net/10138/563824

publishedVersion

Proceedings of

SAT COMPETITION 2023

Solver, Benchmark and Proof Checker Descriptions

Tomáš Balyo, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda *(editors)*

University of Helsinki
Department of Computer Science
Series of Publications B
Report B-2023-1

ISSN 1458-4786
Helsinki 2023

PREFACE

The area of Boolean satisfiability (SAT) solving keeps on making progress. Besides new algorithms and better heuristics, refined implementation techniques turned out to be vital for the success story of SAT solving. To keep up the driving force in improving SAT solvers, SAT solver competitions provide opportunities for solver developers to present their work to a broader audience and to objectively compare the performance of their own solvers with that of other state-of-the-art solvers.

SAT Competition 2023 (SC 2023, `https://satcompetition.github.io/2023/`), a competitive event for SAT solvers, was organized as a satellite event of the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023). SC 2023 stands in the tradition of the previously organized main competitive events for SAT solvers: the SAT Competitions held 2002–2005; biannually during 2007–2013; 2014, 2016–2018, and 2020–2022; the SAT Races held in 2006, 2008, 2010, 2015, and 2019; and SAT Challenge 2012.

SC 2023 consisted of a total of three tracks: Main Track (with CaDiCaL 1.5.3 Hacks and No Limits sub-tracks), Parallel Track and Cloud Track. There were three ways of contributing to SC 2023: by submitting one or more solvers to participate in the competition; by submitting interesting benchmark instances on which the submitted solvers could be evaluated in the competition; and as a new development for 2023 by submitting an unsatisfiability proof checker. An open call for proof checkers resulted in the choice of four proof checkers to choose from for solver developers participating in the competition.
The rules of SC 2023 required all contributors to submit a short, 1–2 page long description as part of their contribution. This compilation contains these non-peer-reviewed descriptions in a single volume, providing a way of consistently citing the individual descriptions and finding out more details on the individual solvers, proof checkers and benchmarks.

Successfully running SC 2023 would not have been possible without active support from the community at large. We would like to thank the StarExec initiative (http://www.starexec.org) for the computing resources needed to run SC 2023. Many thanks go to Aaron Stump for his invaluable help in setting up StarExec to accommodate for the competition's needs. Furthermore, we thank Amazon for providing the resources and support to develop parallel and distributed solvers on the AWS cloud and for executing the Cloud and Parallel tracks. Finally, we would like to emphasize that a competition does not exist without participants: we thank all those who contributed to SC 2023 by submitting either solvers or benchmarks and the related description.

Tomáš Balyo, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, & Martin Suda
SAT Competition 2023 Organizers

Contents

SOLVER DESCRIPTIONS

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Watch Sat and LTO for CaDiCaL

Norbert Manthey
nmanthey@conp-solutions.com
Dresden, Germany

***Abstract*—When reading the source code of the solver CaDiCaL, many differences to solvers based on Minisat 2.2 or Glucose 2.2 can be found. Porting an algorithm detail in unit propagation from CaDiCaL to MergeSAT resulted in a performance degradation. In MergeSAT, when watching a satisfied literal during unit propagation, the clause is moved to the watch list of that literal. In 2021, CaDiCaL just updated the blocking literal of the clause and keeps the clause in the current watch list. Since then, this change was not picked up in CaDiCaL. Hence, we ported MergeSAT's behavior to CaDiCaL. Furthermore, link-time-optimization, as used in MergeSAT, is enabled.**

I. Unit Propagation Improvements

SAT solvers are used in many fields. Hence, some solvers are heavily tuned to perform well for target applications. Other research focusses on improving the overall solver performance in general. Many heuristic and algorithmic extensions to the core algorithm have been proposed [1]. The overall runtime distributions among the algorithm components still did not change significantly: unit propagation still takes a vast majority of the overall runtime [6], [3].

A. Watching Clauses in Propagation

The modification presented in this description alters an implementation detail of unit propagation that is different in CaDiCaL when being compared to other Minisat 2.2-based SAT solvers that participate in competitive events. The two watched literals scheme has been implemented first in [7]. The next major improvement to skip processing clauses early was to move literals, so called *blocking literals*, from the clause into the watch list data structure. Minisat 2.2 2.1 [2] started to use a blocking literal. When propagating a clause, first the truth value of the blocking literal is checked. In case the blocking literal is satisfied, the related clause is known to be satisfied. Therefore, the clause does not have to be processed further. This technique helps to improve the performance of SAT solvers [6].

In Minisat 2.2, the blocking literal of a clause is typically the other watched literal. However, any other literal of the clause could be chosen.

B. How to Handle Satisfied Clauses

When a blocking literal is not satisfied, the clause has to be processed. During this process, each clause of the watch list for the current literal has to be iterated. For each clause, the truth value of all literals has to be checked, in case we find a *conflict clause* or *unit clauses* that force the extension of the current truth assignment. For satisfied clauses, we only need to process the literals until we find a satisfied clauses.

One difference between CaDiCaL and Minisat 2.2 based solvers is the way how they treat these satisfied clauses. Minisat 2.2 based solvers watch the satisfied literal. CaDiCaL skips updating watch lists. Furthermore, CaDiCaL implements further extensions, like memorizing the literal in a clause that was tested when last processing the clause [4].

a) Always Watching the Satisfied Literal: When a satisfied literal is detected in a clause during propagating a literal, the clause is removed from the current watch list. As a next step, solvers append the clauses to the watch list of the satisfied literal. Both operations are constant time, but require accessing the other watch list, which can lead to a cache miss [6] and TLB miss [3]. The watch list of the other literal can be higher in the search tree, so that the clause will be touched less frequent in the remainder of the search. Restarts might reduce the saving, on the other hand solver today use *partial restarts* [9], *chronological backtracking* [8] as well as *trail saving* [5]. All these technique give this saving back partially.

b) Just Updating the Blocking Literal: As an alternative, CaDiCaL keeps watching the current literal, which is now falsified, but updates the blocking literal to the satisfied literal. While this breaks the assumption that falsified literals are only watched for *conflict clauses* or *unit clauses*, we still know that the clause is satisfied. Hence, breaking this assumption does not have consequences. The positive effect is that the clause does not have to be removed from the current watch list. This results in no cache miss, nor a TLB miss. However, when the search progresses, after backtracking, the same clause might need to be processed again. In case the satisfied literal is still satisfied, only the blocking literal has to be processed. Otherwise, backtracking also removed the assignment for the blocking literal, so that the whole clause needs to be processed again.

c) Watching the Satisfied Literal in CaDiCaL*:* Preliminary testing with MergeSAT when just updating the blocking literal of a clause resulted in a performance degradation. Hence, removing this technique for CaDiCaL might result in a performance improvement. The solver CaDiCaL-watch-sat implements this modification.

Not processing a satisfied clause during propagation soon again can result in a different order of propagated literals, as well as different conflicts, and consequently in different heuristic updates and many different follow-up search steps of the solver. Hence, performance differences can not only be attributed to lower or higher compute resource utilization.

II. Generic Improvements

Besides modifying the algorithm directly, other parameters of the environment can be influenced as well. Helping the CPU to access likely-to-be-accessed memory early with *prefetching* [6], as well as using *(transparent) huge pages* to reduce the paging overhead of a program [3] have been discussed already. Another area to investigate is compiler parameters. By default, compilers optimize code per compilation unit, which usually translates to source files. Optimizations across source files, so called *link time optimization* (LTO), has to be enabled explicitly. Besides spotting programming errors during compile time, LTO also allows to improve the performance of a solver slightly. LTO can be enabled by adding `-flto` to the compiler invocation.

This compile time flag has been added to the build files for CaDiCaL.

III. Availability

The source of the modified CaDiCaL is publicly available at https://github.com/conp-solutions/cadical/tree/watch-sat-flto. The used version of the tool is "rel-1.5.3-3-g598343f". This solver has been submitted to the CaDiCaL hack track.

References

[1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*. Amsterdam: IOS Press, 2009.

[2] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT 2003*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Heidelberg: Springer, 2004, pp. 502–518.

[3] J. K. Fichte, N. Manthey, J. Stecklina, and A. Schidler, "Towards faster reasoners by using transparent huge pages," in *Principles and Practice of Constraint Programming*, H. Simonis, Ed. Cham: Springer International Publishing, 2020, pp. 304–322.

[4] I. P. Gent, "Optimal implementation of watched literals and more general techniques," *J. Artif. Intell. Res.*, vol. 48, pp. 231–251, 2013. [Online]. Available: https://doi.org/10.1613/jair.4016

[5] R. Hickey and F. Bacchus, "Trail saving on backtrack," in *Theory and Applications of Satisfiability Testing – SAT 2020*, L. Pulina and M. Seidl, Eds. Cham: Springer International Publishing, 2020, pp. 46–61.

[6] S. Hölldobler, N. Manthey, and A. Saptawijaya, "Improving resource-unaware SAT solvers," ser. LNCS, C. G. Fermüller and A. Voronkov, Eds., vol. 6397. Heidelberg: Springer, 2010, pp. 519–534.

[7] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *DAC 2001*. New York: ACM, 2001, pp. 530–535.

[8] A. Nadel and V. Ryvchin, "Chronological backtracking," in *Theory and Applications of Satisfiability Testing – SAT 2018*, O. Beyersdorff and C. M. Wintersteiger, Eds. Cham: Springer International Publishing, 2018, pp. 111–121.

[9] P. van der Tak, A. Ramos, and M. Heule, "Reusing the assignment trail in cdcl solvers," *JSAT*, vol. 7, no. 4, pp. 133–138, 2011.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Cadical_rel_Scavel and Cadical_rel_1.5.3.Scavel

1st Zhihui Li, 2nd Guanfeng Wu, 3rd Yang Xu,

4th Keming Wang, 5 th Zhiguo Long, 6thZhibin Yu
School of Mathematics
National-Local Joint Engineering Laboratory of System Credibility Automatic Verification, Southwest Jiaotong University
Chengdu, China
lizhihui@swjtu.edu.cn, wgf1024@swjtu.edu.cn,
xuyang@swjtu.edu.cn, kmwang@swjtu.edu.cn,
zhiguolong@swjtu.edu.cn, zbyu@swjtu.edu.cn

Abstract— This document describes Cadical_rel_Scavel and Cadical_rel_1.5.3.Scavel at the SAT Competition 2023.

I. Introduction

The base solvers we used to implement our techniques are Cadical2022 and cadical-rel-1.5.3, obtained from the SAT Competition 2022 [1]. Based on the very competitive solvers, some minor changes mainly include the following technical solutions: Loose Clause Management, Horn, and Core First Unit Propagation by adjusting the 2-watched scheme[3] and adjusting the program flow with the previous centralized technology.

II. ALGORITHM AND IMPLEMENTATION DETAILS

A. Loose Clause Management

The performance of the CDCL solver is closely related to its learned clause database management. Triggering deletion at the right time and preserving high-quality learning clauses as much as possible are two essential aspects of learned clause management. The original intention of our improvement was to keep the initial learned clause deletion strategy and the quality evaluation standard of the learned clause set and relax the original deletion standard appropriately. Based on the learned clause formation process, we define the proportion of original clauses involved in conflict formation in an appropriate range and the empty clauses formed by conflict formation as original clauses, which serve as the basis for us to relax. Some of the previously deleted learned clauses may be avoided or delayed. According to the experimental test, the proportion of original clauses in the formation of participating conflicts is between 0.108 and 0.328. Suppose the empty clause formed by the conflict is the original clause. In that case, it can be considered that the learned clause obtained from the corresponding conflict analysis is more relevant to the original problem formally described by the CNF instances.

B. Horn and Core First Unit Propagation

In a typical CDCL implementation, a data structure called the 2-watched scheme is commonly used because the main function unit Propagation needs to detect unit clauses as efficiently as possible. The horn clause is an essential type of clause that plays a vital role in automatic reasoning, and the Core clause is defined as one with a literal block distance less than or equal to 7. [2] shows that core first unit propagation can improves the performance of the winner of the SAT Competition 2018, MapleLCMDistChronoBT. So we performed this technique on the horn clause by adjusting the 2-watched scheme [3].

C. External Restart Frame

Quick restart technology is very important for solving UNSAT incidents, and as a major technical module of the CDCL framework, restart is triggered multiple times inside the solver function. Before and after the restart, the values of the variables score of the decision branch are not changed; the 2-watched scheme usually presents one of them in order. The clause literals order or the construction of Elements of watches by the previous propagation forms this order. Usually, the order in every literal 2-watches is fixed. We adjust the program flow with an external restart frame to start the solution process several times based on the number of restarts that have occurred by adding an outer loop around the solve function. Before the solve function is called, we increase the decision branch value of an inactive variable, manage the size of three clause sets (core, iter2, and local) boldly and randomize the existing 2-watched Scheme for the adverse effect of the former solution stage on the latter one is eliminated as far as possible.

III. SOLVERS

The Cadical_rel_Scavel in this submission is a small number of modifications of CaDiCaL2022 and cadical-rel-1.5.3 [1] that participated in SAT competition 2022, which implement our techniques of II. A ~C.

The cadical_rel_1.5.3.scavel solver in this submission is a modification of cadical-rel-1.5.3 that participated in SAT competition 2022, which only implements our techniques of II.B.

IV. ACKNOWLEDGMENTS

The author would like to thank the developers of all predecessors of Minisat, CaDiCaL, Kassat, and all the authors who contributed to the modifications that have been integrated. Their solvers are the foundation of our learning and improvement. This work is also supported by the National Natural Science Foundation of China (Grant No: 62106206).

References

[1] SC 2022, https://satcompetition.github.io/2022/.

[2] Jingchao Chen:Core First Unit Propagation.CoRR abs/1907.01192 (2019)

[3] M. Moskewicz, C. Conor, Y. Zhao, L. Zhang and S. Malik, Chaff: Engineering an efficient SAT solver, in Proc. DAC'01 (2001).

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

CaDiCaL_vivinst, IsaSAT, Gimsatul, Kissat, and TabularaSAT Entering the SAT Competition 2023

Armin Biere
University Freiburg
biere@cs.uni-freiburg.de

Mathias Fleury
University Freiburg
fleury@cs.uni-freiburg.de

Florian Pollitt
University Freiburg
pollittf@cs.uni-freiburg.de

This note describes our solvers entering the SAT Competition 2023. To the CaDiCaL hack track we submitted CaDiCaL_vivinst, which combines vivification and variable instantiation. To the main track we submitted the verified solver IsaSAT, a new improved version of Kissat, and the new highly configurable SAT solver TabularaSAT. Our parallel multi-threaded solver Gimsatul went through several optimizations too and was submitted to the parallel track.

I. CaDiCaL_vivinst

This CaDiCaL hack extends version 1.5.3 with a specific form of variable instantiation as part of vivification [1]. It targets removal of literals with few occurrences, which in turn is hoped to allow more variable elimination.

During vivification clauses clauses are considered to be vivified one-by-one. Each such vivification candidate is negated and all its literal are assumed to be false, which is interleaved with standard Boolean constraint propagation (BCP), while ignoring the candidate. Conflict analysis is then used to determine if the clause can be shortened.

Instantiation is another technique implemented in CaDiCaL. It is based on variable instantiation [2] and differs only slightly from vivification, as it also assumes the negation of the literals of the candidate, except for one literal which is assigned to true. If standard BCP derives a conflict, then we can shorten the clause by removing the literal assumed to be true.

To combine both techniques, the last literal in each vivification candidate is assumed in both phases: first as being false for vivification, then as being true for instantiation. In both cases a conflict after propagation might allow to shrink the clause. As our implementation of vivification sorts literals in the candidate by decreasing number of occurrences (to reduce the necessity for backtracking), this form of variable instantiation tries to remove literals that appear less often.

II. IsaSAT

Our verified SAT solver IsaSAT version sc2023 has been submitted to the main track. It is verified using a refinement approach: We start from PCDCL, a combination of CDCL and various rules to enable inprocessing. Then we refine this non-deterministic calculus down to executable code, which is exported and then compiled by the LLVM compiler. Similar to last year, we submitted only the executable code and not the whole Isabelle development. The latter is available at https://bitbucket.org/isafol/isafol/src/sc2023/Weidenbach_Book/ as part of the IsaFoL development.

Compared to last year, we have only implemented and verified forward subsumption by extending PCDCL with strengthening through (self-)subsumption-resolution (SR). Then we refine to check SR for certain candidates – the candidates appear in occurrence lists, but this is only important at the last step of our refinement.

In order to improve performance on satisfiable instances, IsaSAT now uses two different decision heuristics: VMTF in focused mode and ACIDS [3] in stable mode. The latter uses internally pairing heaps: the idea is to average the score with the current conflict count when bumping a literal. To simplify the formalizing, we have not verified rescaling and instead capped our conflict count at uint64_max (afterwards, it is not incremented anymore, meaning that eventually our ACIDS decision heuristic becomes static). We actually intended to go to EVSIDS like most other solvers from the SAT Competition, but the verification effort for the pairing heaps was high enough that we went for the simpler ACIDS for this year's competition (EVSIDS can also use pairing heaps).

We further found and fixed one performance issue, which was due to the (unverified) parser passing clauses to our (verified) solver in an array. Previously we forgot to properly free this array after initializing the solver internal data structures. Fixing this issue is not expected to improve solving speed but might lead to fewer memory-outs.

III. Gimsatul

Our parallel solver Gimsatul was implemented for the last SAT Competition 2022 in a rush within two months and thus was missing several features that might help to improve multi-threaded solving and more importantly also was much slower in single threaded mode than Kissat. Some of these issues have been addressed since then in Version 1.1.1.

To improve memory locality, we replaced opaque watcher pointers by offsets to thread-local watchers pushed on a stack. This indexing restricts the number of watchers to $2^{31} - 1$ instead of using pointers in watch lists, but makes room for blocking literals to speed up propagation. We further allocate space in the watcher structure for directly storing literals of clauses of size 3 and 4, thus avoiding additional memory dereferences for such short but non-binary clauses.

The thread-local pools for sharing clauses are now indexed by the glue of shared clauses which makes sharing more fine grained. Mode switching, rephasing, global simplification, as well as local probing and restarts now all follow the same schedule as in Kissat (and include scaling based on formula size). The variable decision priority queue is also initialized in the same way as in Kissat. Vivification is split into a tier1 phase and tier2 phase and has been optimized as well as clause data-base reduction based on tier information.

We added chronological backtracking, which reduces the number of forced backtracks during importing clauses (particularly units). Importing clauses during vivification was improved in a similar way. Finally we eagerly jump binary reasons during propagation to speed-up conflict analysis for instances with man binary clauses.

IV. Kissat

For the new version 3.1.0 (sc2023) of Kissat submitted to the main track of the SAT Competition 2023 we added back vivification of irredundant clauses compared to last year and also simplified the vivification code. We fixed two heuristic bugs, by avoiding to increase the number of conflicts during vivification, as it is used for scheduling various procedures, as well as initializing used flags of learned clauses correctly.

In last year's light version we already removed hyper binary resolution, which freed up one bit for variable indices. Without hyper binary resolution most clauses are actually irredundant and therefore it further did not make sense to also keep the redundant bit in binary virtual clauses, which we dropped then too. This raises the total number of supported variables to $2^{30} - 1$ (so more than one billion variables).

We also incorporated the ESA idea proposed in the competition last year [4] and schedule bounded variable elimination attempts based on variables scores (EVSIDS and VMTF stamps [3]) and refined it further by taking the difference and not as previously the sum when falling back to the number of positive and negative occurrences of a variable. We also went over SAT sweeping again which improved it slightly.

Experience gained in implementing and optimizing parsing and printing LRAT proofs in lrat-trim helped to improve DIMACS parsing and DRAT printing time for Kissat too.

Finally we eagerly jump binary reason clauses during propagation to reduce the time spent in conflict analysis substantially and total solving time slightly for instances with many binary clauses even though it risks missing unique implication points in the binary implication graph.

V. TabularaSAT

As others, we have been exploring different ways to implement SAT solving in a configurable way, in order to perform experiments which are supposed to shed light on understanding to what extend specific techniques contribute to overall solver performance as well as to ease the process of tuning and extending SAT solvers.

In this regard CaDiCaL (following Lingeling) uses (many) run-time options to achieve configurability even though there are some minor compile-time options (used in the competition) to for instance remove all redundant statistics gathering code. The problem with that run-time approach is that the solver has to include at compile-time all the variability needed to support the various options which poses the substantial risk that features not used in a specific configuration of interest inadvertently incur a non-negligible run-time penalty.

To avoid this risk we also explored the other extreme and only used compile-time options in our didactic SAT solver Satch. This allows dependencies of features to be detected by the compiler (making heavy use of the C preprocessor). However this compile-time approach turned out to produce complex code and was too cumbersome to be maintained in general, e.g., when different configurations should share a certain part of the code, such as allowing to use VMTF as alternative for EVSIDS, either with mode switching, or in focused or stable mode only configurations.

As a compromise, to overcome the problems with both approaches, we developed TabularaSAT, which was submitted to the main track of the competition in version number 1.0.0 (sc2023). The basic idea is that we allow only a small number of different compile-time views on the main source, such as "default" and particularly "vanilla". While "default" is the version submitted to the competition and has all the code of redundant and disabled features removed at compile-time, the "generic" view compiles them in and allows to enable them at run-time (which incurs a performance penalty).

The "vanilla" view tries to mimic MiniSAT, but with the additional "baggage" of the generic (and default) view needed to support full variability in TabularaSAT removed.

Besides these efforts to support improved configurability TabularaSAT reimplements most features of Kissat in a cleaner and easier to understand way (in C++), but without compromising on performance. It does not make use of an embedded SAT solver though (such as Kitten in Kissat) and thus SAT sweeping and semantic gate extraction are missing. On the other hand the implementation of the clause arena and its use is faster while still being cleaner than in Kissat. Performance of TabularaSAT and Kissat are comparable.

References

[1] C. Li, F. Xiao, M. Luo, F. Manyà, Z. Lü, and Y. Li, "Clause vivification by unit propagation in CDCL SAT solvers," *Artif. Intell.*, vol. 279, 2020.

[2] G. Andersson, P. Bjesse, B. Cook, and Z. Hanna, "A proof engine approach to solving combinational design automation problems," in *Proceedings of the 39th Design Automation Conference, DAC 2002, New Orleans, LA, USA, June 10-14, 2002*. ACM, 2002, pp. 725–730. [Online]. Available: https://doi.org/10.1145/513918.514101

[3] A. Biere and A. Fröhlich, "Evaluating CDCL variable scoring schemes," in *SAT*, ser. Lecture Notes in Computer Science, vol. 9340. Springer, 2015, pp. 405–422.

[4] S. Li, J. Coll, C.-M. Li, M. Luo, D. Habet, and F. Manjà, "Solvers Cadical ESA and Kissat MAB ESA in 2022 SAT competition," in *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2022-1. University of Helsinki, 2022.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Kissat_MAB_prop in SAT Competition 2023

Yu Gao
Theory Lab, Central Research Institute, 2012 Labs, Huawei Technologies Co., Ltd.
Beijing, China
gaoyu99@huawei.com

***Abstract*—This document explains the features of our SAT solver Kissat_MAB_prop.**

I. Introduction

Recent research has shown that the Multi-Armed Bandit (MAB) framework efficiently combines different search heuristics for SAT solvers. However, a reward function to estimate the performance of the branching heuristics may prefer the better heuristic for some instances while misleading on the others. We propose a new reward function in the solver Kissat_MAB_prop.

II. Reward Function By Propagations

[1] measures the performance of a heuristic by the estimated fraction of search tree it explores per decision. This naturally translates to

$$reward = \frac{\log_2(propagations)}{decidedVars}$$

in the context of SAT solvers. $propagations$ and $decidedVars$ denote the number of propagations and touched variables in a run of VSIDS or CHB, respectively. Note that we use the number of propagations instead of the number of decisions. We also try to normalize the reward function by time, i.e., let $propagations$ and $decidedVars$ be the number of propagations and touched variables per second in a run.

III. Partial Symmetry Breaking by RAT

[7] proposed a method to express symmetry breaking in DRAT proofs. To break a symmetry σ where $\sigma(x_i) = p_i$ for $1 \leq i \leq n$, they perform the following steps.

1) Introduce a new variable x'_i of each variable x_i in the support of σ. The value of x'_i is equal to x_i if $(x_1, \dots, x_n)$ is lexicographically no more than $(p_1, \dots, p_n)$. Otherwise, the value of x'_i is equal to p_i.
2) For each clause c containing one or more x_i or $\bar{x}_i$, we add a new clause c' that is obtained from c by replacing the $x_i, \bar{x}_i$ respectively by $x'_i, \bar{x'_i}$. Delete c after adding c'.
3) Prove $(x'_1, \dots, x'_n) \leq (p'_1, \dots, p'_n)$. (The $\leq$ means "lexicographically no more than".)

The claim in Item 3 is true for all σ satisfying $\sigma = \sigma^{-1}$. Otherwise, we may need to apply the symmetry σ for more than once to make sure $(x'_1, \dots, x'_n) \leq (p'_1, \dots, p'_n)$. For example, suppose the symmetry σ satisfies $\sigma(x_1) = x_3, \sigma(x_2) = x_1, \sigma(x_3) = x_2$. Also suppose $(x_1, x_2, x_3) = (1, 0, 0)$. We have $(x_1, x_2, x_3) > (p_1, p_2, p_3) = (0, 1, 0)$. Thus, x'_i is equal to p_i. We do not have $(0, 1, 0) = (x'_1, x'_2, x'_3) \leq (p'_1, p'_2, p'_3) = (0, 0, 1)$.

We use bliss [5] to detect symmetry in the formula and implement a method similar to [7] for all symmetries σ satisfying $\sigma = \sigma^{-1}$. In addition to [7], we also performs some additional steps to transfer the lexicographical symmetry breaker back to the original variables x_i:

1) Fill in the proof for the claim $(x'_1, \dots, x'_n) \leq (p'_1, \dots, p'_n)$.
2) Delete all clauses containing the auxiliary variables s_i. Now x_i are not contained in any clause.
3) Add clauses $\neg x_i \vee x'_i$ and $x_i \vee \neg x'_i$. This means that $x_i = x'_i$.
4) Add the original clauses back. Clause c is redundant because of c' and $x_i = x'_i$.
5) Proves the *breaker*, $(x_1, \dots, x_n) \leq (p_1, \dots, p_n)$. Each clause in the breaker is redundant because we proved $(x'_1, \dots, x'_n) \leq (p'_1, \dots, p'_n)$ and have $x_i = x'_i$.
6) Keep the original formula and the proof for $(x_1, \dots, x_n) \leq (p_1, \dots, p_n)$. Delete the other clauses.

In general, this method cannot break all symmetries in the formula. If two symmetries contains a common variable, the second symmetry cannot be broken becuase x_i are contained in the breaker for the first symmetry. This falsifies the claim in Step 2 and invalidates Step 3.

IV. Implementation

Our new solver is based on the Kissat_Pre solver in SAT competition 2022 [2]. The main changes we implement are the following.

- Change the reward function to depend on number of propagations (instead of decisions) in MAB between VSIDS and CHB as mentioned above.
- Bump scores of related variables at the beginning of the search [3] so that VSIDS and CHB do not cold-start.
- Preprocess the formula to generate PR clauses using the PReLearn framework [4]. Our solver calls itself as the inner solver. When called as the inner solver, it skips some preprocessing steps.
- Extract hints from Fourier-Motzkin Variable Elimination [2].
- Add pseudo-boolean proof logging for preprocessing in [2].

- Detect pigeonhole contradiction while preprocessing and prove unsatisfiability by PR.
- Use bliss [5] to detect symmetry in the formula and partially break the symmetries as mentioned above.
- Use 4-nary heap instead of binary heap for maintaining variable scores.
- Add -flto for optimizations while compling.

References

[1] A. Paparrizou, and H. Wattez. Perturbing Branching Heuristics in Constraint Solving. In H. Simonis, editor, Principles and Practice of Constraint Programming, pages 496-C513, Cham, 2020. Springer International Publishing.

[2] Z. Chen, X. Zhang, S. Cai, and P. Lu. CDCL Solvers with Improved Local Search Cooperation and Pre-processing. SAT COMPETITION 2022, pages 37-C38, 2022.

[3] M. Osama, and A. Wijs. Multiple Decision Making in Conflict-Driven Clause Learning, in Proc. of ICTAI (Nov. 2020), Baltimore, USA. IEEE, 2020, pp. 161-C169.

[4] J. E. Reeves, M. J. H. Heule, and R. E. Bryant. Preprocessing of Propagation Redundant Clauses. In Automated Reasoning: 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8–10, 2022, Proceedings. Springer-Verlag, Berlin, Heidelberg, 106–124. https://doi.org/10.1007/978-3-031-10769-6_8

[5] T. Junttila, and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics, pages 135-C149. SIAM, 2007.

[6] R. K. Tchinda, and C. T. Djamegni. PADC MapleLCMDistChronoBT, PADC Maple LCM Dist and PSIDS MapleLCMDistChronoBT in the SR19, SAT RACE 2019, p. 33.

[7] M. J. H. Heule,W. Hunt, and N. Wetzler. Expressing Symmetry Breaking in DRAT Proofs. Proc. of the 25th Int. Conference on Automated Deduction (CADE 2015). Volume 9195 of LNCS., Cham, Springer (2015) 591-606

SBVA-CADICAL and SBVA-KISSAT: Structured Bounded Variable Addition

Andrew Haberlandt* and Harrison Green*
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
{ahaberla, harrisog}@cmu.edu
*Authors contributed equally

***Abstract*—We describe two submissions to the SAT Competition 2023 that use Structured Bounded Variable Addition (SBVA) as a preprocessor to CaDiCaL (SBVA-CADICAL) and Kissat (SBVA-KISSAT).**

I. BOUNDED VARIABLE ADDITION

Bounded Variable Addition (BVA) [1] is a preprocessing technique for re-encoding formulas by adding new auxiliary variables and eliminating clauses. BVA is often able to substantially reduce formula size by identifying sets of clauses which can be generated by a much smaller set resolving on a new auxiliary variable.

BVA uses a greedy algorithm which systematically constructs these sets of clauses by identifying the next best literal at each step. Once the algorithm is not able to add more clauses to the set, a new auxiliary variable is created and the process repeats. For some types of problems, BVA can improve solve time by an order of magnitude, even if the formula size is not significantly reduced.

However, the original implementation of BVA is *fragile* with respect to formula randomization. Many literals may be *tied* in the greedy step, and breaking these ties differently leads to different variable additions. Since the original BVA algorithm implicitly breaks ties using the order of variables in the formula, it produces different auxiliary variables when the formula is scrambled. In practice, we find that scrambling the formula often disrupts the ability of BVA to generate large speedups.

II. STRUCTURED BVA

Structured Bounded Variable Addition (SBVA) [2] improves upon BVA's greedy algorithm with a tiebreaking heuristic based on the formula's Variable Incidence Graph. Since this heuristic does not consider variable order or polarity, it is robust to formula randomization. Furthermore, we find that the variables added by SBVA more closely reflect problem-specific structure even in randomized formulas; in the packing k-coloring problem, for example, SBVA can generate auxiliary variables which cluster nearby variables in the 2D grid space of the problem.

AH is supported by NSF Graduate Research Fellowship Grant No. DGE-2140739.

We applied BVA and SBVA as preprocessors to CaDiCaL in a large scale benchmark and found that even though these preprocessors incur a runtime overhead, the total runtime (preprocessor time + solve time) was *reduced* on average compared to running the original formulas directly in CaDiCaL. Additionally, the variant with SBVA as a preprocessor was able to solve the *most* formulas across every category of the benchmark even though it was provided with randomized instances.

III. IMPLEMENTATION

Our submission to the 2023 SAT competition augments existing solvers CADICAL (version 1.5.3) and KISSAT (version 3.0.0) [3] with SBVA as a pre-processing step. In both configurations, we use SBVA to generate a reduced CNF formula which is then passed directly to CaDiCaL/Kissat. Our implementation of SBVA is capable of producing DRAT proofs for the preprocessing step which we concatenate with the solver's proof to generate a full proof for UNSAT instances.

In some cases, SBVA can impose a large overhead on instances which can be solved quickly without reduction. In a large-scale benchmark performed on more than 29 000 formulas from previous SAT competitions, we found it was effective to impose a timeout policy on SBVA [2]. Specifically, in these configurations we allow SBVA to run for 200 seconds. Upon reaching this timeout, the current formula (with any partial reductions that have been performed so far) is passed to the solver. In our benchmark, we found that SBVA terminates within 200 seconds on approximately 95% of problems.

REFERENCES

[1] N. Manthey, M. J. Heule, and A. Biere, "Automated reencoding of boolean formulas," in *Hardware and Software: Verification and Testing: 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers 8.* Springer, 2013, pp. 102–117.

[2] A. Haberlandt, H. Green, and M. J. Heule, "Effective auxiliary variables via structured reencoding," in-review at SAT 2023.

[3] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Kissat-INCSP: Introducing High Performing Software Prefetching Conscious Kissat-INC

Karthikeya Namoju[1], Kalind Karia[2], Supratik Chakraborty[3], Biswabandan Panda[4]
[1,3,4]Department of Computer Science and Engineering, [2]Department of Electrical Engineering
Indian Institute of Technology, Bombay
karthikeyaiitb@gmail.com[1], kalind1610@gmail.com[2], {supratik[3], biswa[4]}@cse.iitb.ac.in

I. Introduction

Kissat-inc [1] is one of the fastest SAT solvers as per SAT competition 2022. However, we find the performance of kissat-inc can further be improved by making it conscious of some of the optimizations (like better memory layout and software prefetching) from the computer architecture world. In the pursuit of achieving better performance in runtime, we profile kissat-inc with some of the benchmarks provided with SAT 2022. Based on the profiling of a few benchmarks, we find that a certain portions of kissat-inc is *memory-intensive*, which means most of the time the data required for the data structures used in kissat-inc are not available at a multi-level cache hierarchy, which is ubiquitous in modern processors. This memory bottleneck causes the processor to stall for long affecting the overall runtime of the SAT solver. In section II we discuss two major hotspots of the solver and in section III we present our solutions which are based on software prefetching and cache conscious programming. We haven't made any algorithmic changes to the solver. And in section IV we present the results of our experiments. Finally we end this report with section V Conclusion, which highlights some of the algorithmic suggestions we have.

II. The hotspots

`PROPAGATE_LITERAL()` is the top hotspot for most of the benchmarks. The solver spends most of its execution time in this function. Main purpose of it is to traverse the watch list of a literal to find a conflicting clause[1]. Accessing first entry of the watchlist and all the non-binary clauses it encounters while traversing the list are the two major memory bottlenecks. A significant fraction of execution pipeline slots could be stalled due to these demand memory loads.

`kissat_next_decision_variable()` is the next top hotspot for a few benchmarks. The algorithm maintains a queue for a set of *variables* (Each literal corresponds to two variables, `lit` and `not_lit`). The queue is stored in a linked-list `links`. Say only two variables `v1` and `v2` are in the queue then the following holds :

```
links[v1].next = v2;
links[v2].prev = v1;
```

[1]We only talk about the functionality that is relevant to the bottleneck.

When `solver->stable` is set to `false`, this function fetches the last enqueued unassigned variable by traversing the `links` data structure in the queue order. In each iteration, it accesses the entry in `links` and `values` corresponding to a particular variable. Since both the data structures are indexed by variables, there is no spatial locality or a systematic pattern in these accesses hence the architecture couldn't really predict its next access and prefetch it into the cache beforehand. Essentially in every iteration, it has to fetch an entry of this huge list from the main memory, which is proved to be costly [2].

III. Enchancements

A. Cache friendly queue

First, we present our solution to the 2nd hotspot by using an auxiliary data structure. Just for the purpose of this function, we maintain a list of `struct exp_queue` to store the queue explicitly i.e in the queue order. The algorithm is implemented in such a way that this list is always in sync with `links`.

```
struct exp_queue
{ unsigned variable_idx, prev, next; }
```

`dequeue()` operation introduces some garbage entries in this list, hence we still need the fields `prev` and `next`. We dynamically grow this list on-demand and is compacted whenever its size is increased. Now we can traverse the queue efficiently as they are arranged contiguously in the memory. But we still have to deal with accesses of entries in `values`. For this, we have used software prefetching to prefetch the value required in the next iteration. There's some room for improvement here, you can store a copy of the variable's value in the `exp_queue` itself. But this implementation requires more understanding of the code, hence we didn't do it.

With this we were able to get a massive *600 secs* improvement on a benchmark[2].

B. Software prefetching

Now we present our solution to the first hotspot. The core idea is to use software prefetching to make watchlist entries available in the cache by the time the program accesses them. One of the challenges in software prefetching is to prefetch

[2]Its name is ncc_none_7047_6_3_3_0_0_420.cnf. It was used in SAT2022 competition.

the data not too late or not too early, meaning the prefetched data should still be in the cache when it's actually accessed. Also, the memory address of the required data should be accessible while prefetching. We prefetch the head of the watchlist corresponding to the next PROPAGATE_LITERAL function call in the caller as this is the first location in the whole code where we can access its address. Here we make an important observation on the size of watchlists that we encounter in PROPAGATE_LITERAL().

We have taken 266 benchmarks from 2022 SAT competition and plotted watchlist size distribution across these in Fig.1. We

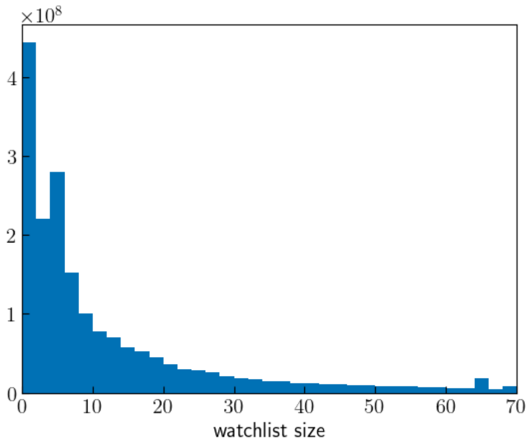


Fig. 1. Histogram of the sizes of watchlists encountered

have observed that *86%* of the time, watchlist size is less than 50, which means the loop in PROPAGATE_LITERAL() will last around 50 iterations. Since each iteration is not memory intensive, it is likely that prefetched data will still be at some cache level when it's actually accessed. Note that there's no point in prefetching watchlist entries corresponding to the next iteration in PROPAGATE_LITERAL() as the watchlist is contiguous. Here we could also prefetch conditionally based on size of the watchlist. We tried prefetching only when size (watches->size) is greater than 0, 2 and 4. But we have observed that this is degrading the performance, most likely because it might have to go to memory to fetch the size value.

Instead of aggressive prefetching (i.e prefetching all the time), we have tried a heuristic that divides the whole execution process into several phases. We used luby sequence to decide the start point of every phase. Every phase can be divided into 2 sub-phases, Exploration and Exploitation phases. In the exploration phase we compute the average load latency to fetch the head of the watchlist and if it is greater than an empirically determined threshold, then we prefetch in the exploitation phase. But we weren't able to get promising results with this because of the overhead incurred by the heuristic (overhead because of all the additional data accesses introduced). This also proves the fact that the hostpot function, PROPAGATE_LITERAL is highly sensitive to load instructions. Moreover, we tried switching off the prefetching for the whole run and keeping all the code corresponding to the heuristic inplace. We observed that there was a substantial degradation in performance with respect to the case where we do prefetching based on exploration phase. This means the heuristic is working as expected but the overhead is masking all the improvement we got.

IV. Experiments

We ran *kissat-incsp* on the starexec cluster against 266 benchmarks that were used in SAT 2022 competition. We were able to get a total of 4196 seconds improvement over *kissat-inc*. All these 266 benchmarks did not timeout in SAT 2022 competition. We have sorted benchmarks based on the improvement we got with respect to *kissat-inc* and plotted it for the top 50 benchmarks in Fig. 2.

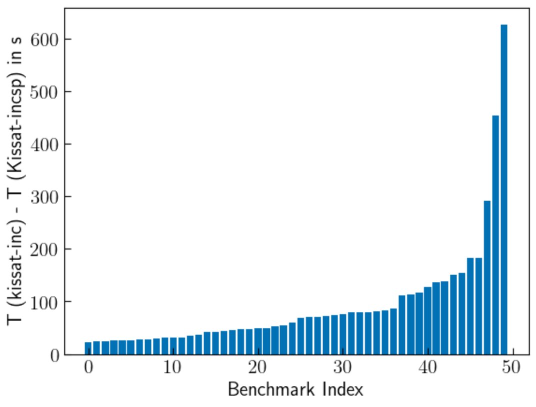


Fig. 2. Enchancement of top 50 benchmarks (top with respect to the total decrease in execution time wrt *kissat-inc*).

V. Conclusion

There's still a lot of scope for improvement from computer architecture point of view. We were able to get a substantial enhancement with just aggressive prefetching, so we expect certain algorithmic changes can bring in a huge improvement. One of them could be traversing non-binary and binary clauses in two separate phases instead of one and maintaining spatial locality for non-binary clauses in the main memory (in the order of accesses in PROPAGATE_LITERAL). Since accessing non-binary clauses is one of the major hotspots, we expect the solver to perform better if all the clauses are accessed in a contiguous fashion.

References

[1] Z. Chen, X. Zhang, S. Cai, and P. Lu. CDCL Solvers with Improved Local Search Cooperation and Pre-processing. Proceedings of SAT competition 2022, pages 37–38. University of Helsinki, 2022.

[2] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. Proceedings of SAT competition 2020, pages 50–53. University of Helsinki, 2020.

SEQFROST at the SAT Competition 2023

Muhammad Osama and Anton Wijs
Department of Mathematics and Computer Science
Eindhoven University of Technology, Eindhoven, The Netherlands
{o.m.m.muhammad, a.j.wijs}@tue.nl

I. INTRODUCTION

This paper presents a brief description of our solver SEQFROST which stands for *Sequential Formal ReasOning about SaTisfiability* in 3 different configurations. SEQFROST was a new solver first introduced in our last year submission [1] with efficient data structures and many code optimizations. In this year submission, we updated our sorting strategies and overhauled most of the data structure names to improve readability. We also fixed a slight bug in SEQFROST parser that caused empty clauses in the input formula to be skipped. Further, as in instead of resetting the number of target assignments, we decay by some factor. The core of SEQFROST heuristics and its simplifications that are based on our work in [2]–[5] remains unchanged.

II. DECISION MAKING

The decision-making step in SEQFROST switches periodically from the standard single-decision procedure as originally introduced in CDCL search to our MDM procedure previously presented in [6]. Both single and multiple decisions are chosen according to VSIDS, VMTF, and CHB [7] branching heuristics. Last year, we added the latter to our solver decision heuristics to improve the quality of the picked decisions in MDM. SEQFROST decides whether to use VSIDS or CHB based on MAB restarts [8]. The decision phases of multiple decisions are still improved via local search but only once at the initial MDM call.

III. VARIABLE ELIMINATION

In gate-equivalence reasoning, we substitute eliminated variables with deduced logical equivalent expressions. Combining gate equivalence reasoning with the resolution rule tends to result in smaller formulas compared to only applying the resolution rule [2]–[5], [9]. Let G_ℓ be the gate clauses having ℓ as the gate output and H_ℓ the non-gate clauses, i.e., clauses not contributing to the gate itself. For regular gates (e.g. AND), substitution can be performed by resolving non-gate with gate clauses as follows: $R_x = \{\{G_x \otimes H_{\neg x}\}, \{G_{\neg x} \otimes H_x\}\}$, omitting the tautological and the redundant parts $\{G_x \otimes G_{\neg x}\}$ and $\{H_x \otimes H_{\neg x}\}$, respectively [10].

In this submission, we focus on finding definitions for irregular gates by checking the unsatisfiability of the *co-factors* formula $\{\mathcal{S}_x|_{\neg x} \cup \mathcal{S}_{\neg x}|_x\}$, that is, the formula obtained by removing all occurrences of x from $\mathcal{S}_x$ and $\neg x$ from $\mathcal{S}_{\neg x}$. In [11], a BDD-based approach is used to solve the co-factors. In our recent work [5], we replace the BDD structure with a function table (bit-vector) encoding the clausal core of the co-factors. The clausal core is mapped back to the original gate clauses G_x and $G_{\neg x}$ by adding back x and $\neg x$, respectively. Then, the set of resolvents $R_x = \mathcal{S}_x \otimes \mathcal{S}_{\neg x}$ is reduced to $\{\{G_x \otimes G_{\neg x}\}, \{G_x \otimes H_{\neg x}\}, \{G_{\neg x} \otimes H_x\}\}$, dropping the redundant part $\{H_x \otimes H_{\neg x}\}$. In contrast to gate substitution, the resolvents $\{G_x \otimes G_{\neg x}\}$ are not necessarily tautological. Function tables are implemented and enabled by default in SEQFROST.

IV. EAGER REDUNDANCY ELIMINATION

ERE was designed originally to target and remove redundant equivalences after a resolution step. It repeats the following until a fixpoint has been reached: for a given formula $\mathcal{S}$ and clauses $C_1 \in \mathcal{S}, C_2 \in \mathcal{S}$ with $x \in C_1$ and $\bar{x} \in C_2$ for some variable x, if there exists a clause $C \in \mathcal{S}$ for which $C \equiv C_1 \otimes_x C_2$, then let $\mathcal{S} := \mathcal{S} \setminus \{C\}$ iff (C is $learnt$ $\vee$ (C_1 is $original \wedge C_2$ is $original$)). The clause C in this case is called a *redundancy* and can be removed without altering the original satisfiability. In addition to the redundancies removal, we observed that if the resolvent $C_1 \otimes_x C_2$ is not equivalent to any clause, it can still subsume many others in $\mathcal{S}$. However, to preserve correctness, subsumed clauses are only strengthened via the generated resolvents. Suppose that $C = (C' \cup C'')$. Extended-ERE (i.e. as we call it in SEQFROST) may strengthen C by removing the redundant literals C' (resp. C'') if $C'' = C_1 \otimes_x C_2$ (resp. $C' = C_1 \otimes_x C_2$).

V. CODE OPTIMIZATIONS

As mention earlier in the introduction section, all pointers of vector-type variables are prefetched to save the time spent in calling the overloaded indexing operator `[]`. Additionally, all functions repeatedly called in unit propagation and conflict analysis are replaced with macros as inlining is not always guaranteed by the compiler. Lastly, the bytes generated by DRAT proof are now stored in a 1-MB buffer. Once, the buffer is full, the data is written to the output file via a single call to `fwrite` (i.e. writes data in burst mode). Compared to previous submissions and other solvers, `putc_unlock` was being called to write on disk byte by byte which, of course, adds unnessary overhead to the proof generation.

VI. SUBMISSIONS

Similar to 2022's submission [1], the solver instance SEQFROST comprises all configurations described in the previous sections, in which MDM with local search, CHB decision heuristic, and all simplifications are enabled with Extended ERE to strengthen original clauses only (e.g. the option `redundancyextend=1` is set). The second configuration SEQFROST-ERE-ALL extends ERE with both original and learnt clause strengthening (e.g. `redundancyextend=2`). The third configuration SEQFROST-NO-EXTEND disables Extended ERE (e.g. `redundancyextend=0`). The initial settings of the SEQFROST have been tuned and tested on TU/e HPC.

REFERENCES

[1] M. Osama and A. Wijs, "SeqFROST at the SAT Race 2022," in *Proc. of SC (2022)*, ser. Report Series B, vol. B-2022-1. University of Helsinki, 2022, pp. 32–34.

[2] M. Osama, A. Wijs, and A. Biere, "SAT Solving with GPU Accelerated Inprocessing," in *Proc. of TACAS (Mar. 2021), Luxembourg*, ser. LNCS, vol. 12651. Springer, 2021, pp. 133–151.

[3] M. Osama and A. Wijs, "Parallel SAT Simplification on GPU Architectures," in *Proc. of TACAS (Apr. 2019), Prague, Czech Republic*, ser. LNCS, vol. 11427. Springer, 2019, pp. 21–40.

[4] ——, "SIGmA: GPU Accelerated Simplification of SAT Formulas," in *Proc. of IFM (Dec. 2019), Bergen, Norway*, ser. LNCS, vol. 11918. Springer, 2019, pp. 514–522.

[5] M. Osama, A. Wijs, and A. Biere, "Certified SAT Solving with GPU Accelerated Inprocessing," *Formal Methods in System Design, Springer*, 2023, accepted.

[6] M. Osama and A. Wijs, "Multiple Decision Making in Conflict-Driven Clause Learning," in *Proc. of ICTAI (Nov. 2020), Baltimore, USA*. IEEE, 2020, pp. 161–169.

[7] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Exponential recency weighted average branching heuristic for sat solvers," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, p. 3434–3440.

[8] M. S. Cherif, D. Habet, and C. Terrioux, "Combining VSIDS and CHB Using Restarts in SAT," in *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), L. D. Michel, Ed., vol. 210. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 20:1–20:19.

[9] M. Osama and A. Wijs, "GPU Acceleration of Bounded Model Checking with ParaFROST," in *Proc. of CAV (Jul. 2021), USA*, ser. LNCS, vol. 12760. Springer, 2021, pp. 447–460.

[10] M. Järvisalo, M. Heule, and A. Biere, "Inprocessing Rules," in *Proc. of IJCAR (Jun. 2012), Manchester, UK*, ser. LNCS, vol. 7364. Springer, 2012, pp. 355–370.

[11] A. Biere, "Lingeling, Plingeling and Treengeling Entering the Sat Competition 2013," in *Proc. of SC (2013)*, ser. Report Series B, vol. B-2013-1. University of Helsinki, 2013, pp. 51–52. [Online]. Available: http://hdl.handle.net/10138/40026

[12] H. E. Bal, D. H. J. Epema, C. de Laat, R. van Nieuwpoort, J. W. Romein, F. J. Seinstra, C. Snoek, and H. A. G. Wijshoff, "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term," *Computer*, vol. 49, no. 5, pp. 54–63, 2016.

Preprocessors PReLearn and ReEncode Entering the SAT Competition 2023

Joseph E. Reeves and Randal E. Bryant
Carnegie Mellon University, Pittsburgh, United States

Abstract—**This system description presents two preprocessors, and four total solving configurations. Two of the configurations use the preprocessor PReLearn that adds short PR clauses to a formula. The derivation produced by PReLearn requires a PR checker. The other two configurations rely on a preprocessor that extracts cardinality constraints from a formula. Then ReEncode rewrites the cardinality constraints with the commander encoding. A DRAT derivation of the rewritten constraints is generated and appended to a SAT solver's proof. After preprocessing, the new formula is passed to the unmodified version 3.0.0 of Kissat for solving.**

PReLearn

PReLearn is a preprocessor that detects short PR (propagation redundant) [1] clauses in a formula. These clauses are added to the formula, and then the formula is passed to the SAT solver Kissat [2]. Short PR clauses can resemble symmetry-breaking used to make exponentially hard problems for resolution easier to solver [3].

PReLearn is built on top of the SAT solver SaDiCaL. The tool iterates over candidate clauses, and for each it generates and solves the *positive reduct*. If the positive reduct is satisfiable, the candidate clause is a PR clause and can be added to the formula. Further, the satisfying assignment for the positive reduct is the witness for the PR clause, and is added to the PR proof generated by PReLearn. There are additional heuristics used for generating candidate PR clauses. We generate only binary candidate PR clauses with the heuristics described in the corresponding paper for PReLearn [4].

After PR clauses are added to the formula, the new formula is passed to the SAT solver Kissat. If the formula is satisfiable, the assignment produced by Kissat will satisfy the original formula. If the formula is unsatisfiable, the DRAT proof produced by Kissat is appended to the PR proof produced by PReLearn. Together these form a complete DPR proof for the original formula.

In one configuration, we run PReLearn for a 100 second timeout, with the default configuration form the paper. In the other configuration "-tern", we run PReLearn for a 300 second timeout and additionally learn ternary PR clauses with an increased depth parameter testing more candidates. We do not run the preprocessor on formulas with more than 500,000 variables or 1 million clauses. PReLearn can be found at https://github.com/jreeves3/PReLearn.

Cardinality Constraint Extraction

A cardinality constraint on Boolean variables has the form $\ell_1 + \ell_2 \cdots + \ell_s \geq k$ and is satisfied by a partial assignment if the sum of the assigned literals is at least k. A commonly occurring cardinality constraint in SAT problems is the at-most-one (AMO) constraint, where at most a single literal in the constraint can be true. There are several ways to encode AMO constraints into conjunctive normal form (CNF), including the pairwise encoding, the Sinz encoding [5], and the commander encoding [6].

We built a tool that detects AMO constraints. For pairwise AMO constraints, the tool greedily expands cliques formed by binary clauses. For non-pairwise AMO constraints the tool guesses problem variables (variables occurring in the constraint) and auxiliary variables (new variables used to encode the constraint) and a set of clauses, then uses a BDD to verify these guesses represent an actual cardinality constraint. In practice, the tool works well at finding Sinz and commander AMO constraints.

Once the constraints are extracted, we use a reencoding scheme. ReEncode takes the detected AMO constraints and reencodes them with the commander encoding. This is effective for formulas with large AMO constraints using the pairwise encoding, since the commander encoding introduces auxiliary variables that often improve solver performance. The reencoded formula is passed to Kissat. The DRAT proof produced by Kissat is appended to a DRAT derivation of the reencoded constraints, forming a complete proof for the original formula. If the formula is satisfiable, the satisfying assignment produced by Kissat must be extended to account for any auxiliary variables that were removed from the original formula during the reencoding procedure.

In the first configuration, we run the cardinality extractor for 300 seconds for pairwise AMO constraints and 300 seconds for non-pairwise AMO constraints, with additional heuristics for terminating early when few constraints are being detected. In the second configuration '-pair', we run a version of the cardinality extractor optimized for extracting pairwise encodings only for 300 seconds. Cardinality extraction tools and cardinality-CaDiCaL can be found at https://github.com/jreeves3/Cardinality-CDCL.

References

[1] M. J. H. Heule, B. Kiesl, M. Seidl, and A. Biere, "PRuning through satisfaction," in *Haifa Verification Conference (HVC)*, ser. LNCS, vol.

10629, 2017, pp. 179–194.

[2] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, “CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020,” 2020, unpublished.

[3] M. J. H. Heule, B. Kiesl, and A. Biere, “Short proofs without new variables,” in *Conference on Automated Deduction (CADE)*, ser. LNCS, vol. 10395. Cham: Springer, 2017, pp. 130–147.

[4] J. E. Reeves, M. J. H. Heule, and R. E. Bryant, *Preprocessing of Propagation Redundant Clauses*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 106–124.

[5] C. Sinz, “Towards an optimal CNF encoding of Boolean cardinality constraints,” in *Principles and Practice of Constraint Programming (CP)*, ser. LNCS, vol. 3709, 2005, pp. 827–831.

[6] W. Klieber and G. Kwon, “Efficient cnf encoding for selecting 1 from n objects,” in *Constraints in Formal Verification (CFV)*, 2007, p. 39.

BREAKID-KISSAT in SAT Competition 2023 (System Description)

Bart Bogaerts
Vrije Universiteit Brussel
Brussels, Belgium
ORCID: 0000-0003-3460-4251

Jakob Nordström
University of Copenhagen
Copenhagen, Denmark
and *Lund University*
Lund, Sweden
ORCID: 0000-0002-2700-4285

Andy Oertel
Lund University
Lund, Sweden
ORCID: 0000-0001-9783-6768

Çağrı Uluç Yıldırımoğlu
Vrije Universiteit Brussel
Brussels, Belgium
e-mail:
cagri.uluc.yildirimoglu@vub.be

***Abstract*—BREAKID-KISSAT combines the symmetry breaking preprocessor BREAKID with the SAT solver KISSAT.**

I. INTRODUCTION

For several years, participation in the main tracks of the SAT competition has required solvers to output proofs in the DRAT format [12]. For a long time, this meant that several state-of-the-art solving techniques are de facto excluded from participation in these tracks. One prime example of such a technique is *symmetry breaking*: while for limited types of symmetries, breaking constraints can be derived in DRAT [9], for the general case, no techniques are known.

In 2023, for the first time, verification in other proof formats is also allowed, thereby allowing also other solving methods into the competition. Our solver is a combination of the symmetry breaker BREAKID [4] and the SAT solver KISSAT [11]. BREAKID-KISSAT outputs produces UNSAT certificates in the VERIPB format and can be verified the VERIPB-CAKEPB pipeline [2].

VERIPB [5]–[8] was originally designed as a proof checker for pseudo-Boolean satisfiability and was recently extended to *pseudo-Boolean optimization* [1], making it not just a viable candidate for certification of SAT techniques, but also for MaxSAT. The underlying proof format is a strict generalization of DRAT. Moreover, since it is based on the cutting planes proof system [3], it also naturally facilitates proof logging for advanced techniques such as XOR and cardinality reasoning [8].

II. MAIN TECHNIQUES

The workflow of our solver is as follows:

- First, the instance to a colored graph in such a way that syntactic symmetries of the problem correspond to automorphisms of the graph.
- Next, BREAKID uses SAUCY [10] to detect automorphisms of the constructed graph.
- Next, BREAKID optimizes the set of detected symmetries to ensure complete breaking of certain subgroups. For each of the resulting symmetries, it creates symmetry breaking clauses.
- Subsequently, the original instance, together with the symmetries is passed to KISSAT, which solves the resulting instance.

BREAKID and KISSAT each produce a part of the resulting proof.

III. AVAILABILITY

The source code of BREAKID is available at https://bitbucket.org/krr/breakid/src. Our modified version of KISSAT to output proofs in the VERIPB format rather than DRAT is available at https://gitlab.ai.vub.ac.be/cagri/kissat3-pb2.

IV. ACKNOWLEDGEMENT

We would like to thank everyone who contributed to SAUCY, BREAKID, and KISSAT for their efforts and for making their tools publicly available.

REFERENCES

[1] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Proceedings of AAAI*, 2022. accepted.

[2] Bart Bogaerts, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. Documentation of VeriPB and CakePB for the SAT competition 2023. Available at https://satcompetition.github.io/2023/checkers.html, March 2023.

[3] William J. Cook, Collette R. Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987.

[4] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 104–122. Springer, 2016.

[5] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020.

[6] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.

[7] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.

[8] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

[9] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In *Proceedings of the 25th International Conference on Automated Deduction (CADE-25)*, volume 9195 of *Lecture Notes in Computer Science*, pages 591–606. Springer, August 2015.

[10] Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Symmetry and satisfiability: An update. In Ofer Strichman and Stefan Szeider, editors, *SAT*, volume 6175 of *LNCS*, pages 113–127. Springer, 2010.

[11] Kissat SAT solver. http://fmv.jku.at/kissat/.

[12] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th Internatjuional Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

New Rephasing Strategies and Their Combinations

Jiongzhi Zheng[1,2] Mingming Jin[1,2] Kun He[1,2] Zhuo Chen[1,2] Jinghui Xue[1,2]
[1]School of Computer Science and Technology, Huazhong University of Science and Technology, China
[2]Hopcroft Center on Computing Science, Huazhong University of Science and Technology, China
{jzzheng,mingmingk,brooklet60,ciaozer,jh_xue}@hust.edu.cn

***Abstract*—This document describes our four SAT solvers, Kissat_MAB_Conflict, Kissat_MAB_Conflict+, Kissat_MAB_DeepWalk+, and Kissat_MAB_Rephases, submitted to the main track of SAT Competition 2023.**

I. Introduction

After selecting the branching variable, modern CDCL SAT solvers usually need to decide the *phase* of the variable, i.e., assigning it 1 or 0. Deciding *phases* is very important, which directly and significantly influences the solvers' performance. Maintaining saved *phases* to decide variables' *phases* is a popular and effective method. Kissat designs six *rephasing* strategies to reset the saved *phases* [1] as follows.

- Original (O), which sets all saved *phases* to 1.
- Inverted (I), which sets all saved *phases* to 0.
- Best (B), which sets saved *phases* to best *phases* if any, otherwise keeps the saved *phases*.
- Walk (W), which changes saved *phases* according to local search results.
- Random ($\#$), which randomly decides the saved *phases*.
- Flipped (F), which flips each saved *phases*.

In Kissat, the default *rephasing* policy is $\{OI(BWOBWIBW\#BWF)^{\omega}\}$ (Original, Inverted; then Best, Walk, Original, Best, Walk, Inverted, Best, Walk, Random, Best, Walk, Flipped is repeated). We propose two new *rephasing* strategies, Conflict (C) and DeepWalk (D), and combine them with the existing strategies to obtain three new *rephasing* policies. We replace the *rephasing* policy in solver Kissat_MAB [2] with the three new *rephasing* policies and obtain three solvers, Kissat_MAB_Conflict, Kissat_MAB_Conflict+, and Kissat_MAB_DeepWalk+, and finally combine several *rephasing* policies to obtain solver Kissat_MAB_Rephases.

II. New *Rephasing* Strategies

- Conflict (C)
  We maintain a conflict *phase* for each variable. Once a conflict clause is found, the *phases* of the variables in the conflict clause will be used to update these variables' conflict *phases*. The Conflict strategy sets saved *phases* to conflict *phases* if any, otherwise keeps the saved *phases*. The conflict *phases* may lead the solvers to detect more unsat cores and learn more clauses.
- DeepWalk (D)
  Inspired by solver Kissat_MAB-HyWalk [3], we design a DeepWalk strategy that sets the number of local search rounds to 5 times the default one.

- The first three authors contribute equally.

III. Kissat_MAB_Conflict

Kissat_MAB_Conflict replaces the *rephasing* policy of Kissat_MAB with $\{OI(BWOBWIBW\#BWFC)^{\omega}\}$.

IV. Kissat_MAB_Conflict+

We find that in Kissat, there is a local optima escaping strategy before each Best strategy, i.e., Original, Inverted, Random, and Flipped. We believe that combining the walking *phases* and the best *phases* in the saved *phases* may help the solvers find models of SAT instances faster. Therefore, Kissat_MAB_Conflict+ replaces the *rephasing* policy of Kissat_MAB with $\{OI(BWBOBWBIBWB\#BWFC)^{\omega}\}$.

V. Kissat_MAB_DeepWalk+

Kissat_MAB_DeepWalk+ replaces the *rephasing* policy of Kissat_MAB with $\{OI(BWBOBWBIBWB\#BDBF)^{\omega}\}$.

VI. Kissat_MAB_Rephases

Kissat_MAB_Rephases combines the *rephasing* policies in Kissat_MAB, Kissat_MAB_Conflict, Kissat_MAB_Conflict+, Kissat_MAB_DeepWalk+, and $\{OI(BWBOBWIBW\#BWFC)^{\omega}\}$.

References

[1] A. Biere, M. Fleury, "Chasing Target Phases," Workshop on the Pragmatics of SAT, 2020.
[2] M. S. Cherif, D. Habet, C. Terrioux, "Kissat MAB: Combining VSIDS and CHB through Multi-Armed Bandit," SAT COMPETITION 2021, 2021: 15.
[3] J. Zheng, K. He, Z Chen, J. Zhou, C. M. Li, "Combining Hybrid Walking Strategy with Kissat MAB, CaDiCaL, and LStech-Maple," SAT COMPETITION 2022, 2022: 20.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

`kissat-hywalk-gb`, `kissat-hywalk-exp`, `kissat-hywalk-exp-gb`, and `malloblin` Entering the SAT Competition-2023

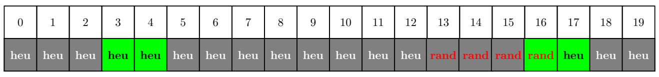
Md Solimul Chowdhury
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
mdsolimc@cs.cmu.edu

Abstract—**This document describes 4 SAT solvers: `kissat-hywalk-exp` `kissat-hywalk-gb`, `kissat-hywalk-exp-gb`, and `malloblin`, which are entering to the SAT Competition-2023. The first three solvers are submitted to the maintrack of the competition and based on the following 2 ideas: 1) Bounded randomized exploration amid conflict depression phases and 2) Activity score bumping of variables that appear in the glue clauses. `malloblin` is a portfolio-based distributed SAT solver that includes a new stochastic local search (SLS) solver `yallin` in its employable solvers collection.**

I. Bounded Exploration amid a CD Phase

This approach is based on our observation that search in Conflict Directed Clause Learning (CDCL) entails clear patterns of bursts of conflicts followed by longer phases of *conflict depression (CD)* [1]. During a CD phase, for a consecutive number of decisions, a CDCL solver is unable to generate conflicts, from which the search could learn clauses to prune the search space. To correct the course of such a search, we propose to use random exploration to combat conflict depression. In this approach, when the search enters into a *substantially long CD phase*, instead of using the currently active decision heuristic, we employ a uniform random strategy for selecting decision variables. The goal of this random exploration is to find conflicts amid a substantially long CD phase, in which the currently active decision heuristic is unable to find a conflict. This random selection continues, until the search finds a conflict or takes a maximum of $s > 0$ random steps. We call this approach *depth bounded* (DB) exploration.

Fig. 1 shows how this approach works.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
heu	heu	heu	🟩 heu	🟩 heu	heu	heu	heu	heu	heu	heu	heu	heu	rand	rand	rand	🟩 rand	🟩 heu	heu	heu

Fig. 1: Assume that a CDCL solver is running a given instance. This figure shows 20 consecutive decisions taken by that solver. The top row shows decision indexes which starts at 0 and ends at 19. In the second row, the grey cells depict decisions with no conflict and green cells depict decisions with non-zero conflicts. For a decision, the text inside a colored cell of the bottom row denotes type of decision strategy (**`heu`**: heuristic decision, **`rand`**: random decision) used at that decision. In this search snapshot, a long CD phase starts at the 5th decision. Amid this long CD phase, the search decides to perform random decisions from decision 13. At decision 16, with a random decision the search finds a conflict. This results in the end of the current CD phase.

II. Glue Variable Bumping

Let a CDCL SAT solver M is running a given SAT instance $\mathcal{F}$ and the current state of the search is S. We call the variables that appeared in at least one glue clause up to the current state S *Glue Variables*. We design a structure-aware variable score bumping method named *Glue Bumping* (GB) [2], based on the notion of *glue centrality (gc)* of glue variables. Given a glue variable v_g, glue centrality of v_g dynamically measures the fraction of the glue clauses in which v_g appears, until the current state of the search. Mathematically, the glue centrality of v_g, $gc(v_g)$ is defined as follows:

$$gc(v_g) \leftarrow \frac{gl(v_g)}{ng}$$

, where ng is the total number of glue clauses generated by the search so far. $gl(v_g)$ is the glue level of v_g, a count of glue clauses in which v_g appears, with $gl(v_g) \leq ng$.

A. The GB Method

The GB method modifies a CDCL SAT solver M by adding two procedures to it, named *Increase Glue Level* and *Bump Glue Variable*, which are called at different states of the search. We denote by M^{gb} the GB extension of the solver M.

Bump Glue Variable: This procedure bumps a glue variable v_g, which has just been unassigned by backtracking. First a bumping *factor (bf)* is computed as follows:

$$bf \leftarrow activity(v_g) * gc(v_g)$$

, where $activity(v_g)$ is the current activity score of the variable v_g and $gc(v_g)$ is the glue centrality of v_g. Finally, the activity score of v_g, $activity(v_g)$ is bumped as follows:

$$activity(v_g) \leftarrow activity(v_g) + bf$$

III. A Linear Weight Transferring Algorithm for SLS

The weight transferring algorithm Divide and Distribute Fixed Weights (DDFW) algorithm [6] transfers fixed amount of wights from satisfied clauses to falsified clauses at a local minima, as an escape measure from local minimas. We conjectured that the fixed weight transfer in DDFW is a contrived design choice, and a dynamic weight transfer rule would be more natural, and be helpful for faster escape from local minimas. Therefore, our new SLS solver yallin [4] (built on top the SLS solver yalsat [3]) introduces a linear weight transfer rule, which transfers a dynamic amount of weights from satisfied clauses to the falsified clauses, whenever the search encounters a local minima. For a given falsified clause C_f and satisfied clauses C_s, this new weight transfer rule transfers

$$\mathtt{a} * W(C_s) + \mathtt{c}$$

amounts of weights to C_f, where $W[C_s]$ is the current weight of the clause C_s, $0 \leq a \leq 1$ is a floating-point constant, and $c \geq 1$ is an integer constant. Our initial experiments showed that yallin improves over yalsat, on top of which yallin is built on.

IV. Solvers Description

We have submitted three CDCL SAT solvers to the main-track of the SAT Competition-2023, which are based on combinations of the two approaches described in the section I and II. The fourth solver is a portfolio-based distributed solver that uses the technique presented in section III. In the following, we describe our solvers:

a) ***kissat-hywalk-gb:*** This solver implements the GB method on top of kissat_MAB_HyWalk [7], the winner of the main-track of SAT Competition-2023. kissat_MAB_HyWalk employs three branching heuristics: VSIDS, CHB and VMTF. In kissat-hywalk-gb, the GB scheme is kept active only when VSIDS and CHB are active.

b) ***kissat-hywalk-exp:*** The solver kissat-hywalk-exp implements the DB strategy on top of kissat_MAB_HyWalk, only when VSIDS and CHB are active.

c) ***kissat-hywalk-exp-gb:*** This solver implements both DB and GB technique on top of kissat_MAB_HyWalk, only when VSIDS and CHB are active.

d) ***malloblin:*** In malloblin, we have replaced yalsat with yallin in mallob[5], the winner in the cloud-track of the SAT Competition-2022.

References

[1] Md Solimul Chowdhury and Martin Müller and Jia You, Guiding CDCL SAT Search via Random Exploration amid Conflict Depression. In Proceedings of AAAI-2020:1428-1435.

[2] Md. Solimul Chowdhury, Martin Müller, Jia-Huai You, Exploiting Glue Clauses to Design Effective CDCL Branching Heuristics. In Proceedings of CP 2019: 126-143.

[3] yalsat, https://fmv.jku.at/yalsat/, access date: 15-April-2023.

[4] yallin, , https://github.com/solimul/yal-lin, access date: 15-April-2023.

[5] mallob, , https://github.com/domschrei/mallob, access date: 15-April-2023.

[6] Abdelraouf Ishtaiwi, John Thornton, Abdul Sattar, Duc Nghia Pham: Neighbourhood Clause Weight Redistribution in Local Search for SAT. CP 2005: 772-776

[7] Jiongzhi Zheng, Kun He1 Zhuo Chen, Jianrong Zhou, and Chu-Min Li. Combining Hybrid Walking Strategy with Kissat MAB, CaDiCaL, and LStech-Maple. Proceedings of SAT Competition-2022:20-21.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

MapleCaDiCaL

Jonathan Chung
University of Waterloo
Waterloo, Canada
ORCID: 0000-0001-5378-1136

Sam Buss
UC San Diego
La Jolla, United States of America
ORCID: 0000-0003-3837-334X

Vijay Ganesh
University of Waterloo
Waterloo, Canada
ORCID: 0000-0002-6029-2047

***Abstract*—The MapleCaDiCaL solvers combine the CaDiCaL SAT solver with the idea of priority-based Boolean Constraint Propagation (BCP). MapleCaDiCaL implements both the Immediate BCP and Delayed BCP modes, and uses a Multi-Armed Bandit (MAB) framework with Thompson sampling to switch adaptively between them. We submit four different configurations of the solver.**

***Index Terms*—Boolean Constraint Propagation, Multi-Armed Bandit, Thompson Sampling**

I. Priority-Based BCP

Priority-based BCP [3] modifies the orders in which variables are assigned and propagated during unit propagation in a CDCL SAT solver. This affects the conflicts detected by the BCP algorithm, which changes the clauses learnt by the solver for a given conflicting partial assignment. Since Delayed BCP empirically outperforms Out-of-Order BCP [3], we choose to implement Delayed BCP and not Out-of-Order BCP.

An algorithmic view of the main Delayed BCP procedure is presented as Algorithm 1. The Delayed BCP algorithm modifies the traditional BCP algorithm to respect a priority order by delaying the assignment of a variable until the point at which BCP processes that variable (delayed variable assignment), propagating variables in the order that they are assigned. This choice delays the detection of conflicts that would otherwise have been found under immediate variable assignment, but ensures that the propagation order is still a topological order with respect to the implication graph. It is only after a variable has been assigned a value that it can contribute to unit propagation.

Since variable assignment is postponed, Delayed BCP must check the propagation queue when searching for conflicts and before adding a variable to the queue. Additionally, when a variable occurs with one polarity in the propagation queue, but is queued with the opposite polarity by some other propagation, the conflicting literal must first be assigned and placed on the assignment trail before the BCP algorithm reports a conflict. This is necessary to maintain the invariant on the assignment trail expected by the conflict analysis procedure.

To minimize the introduction of additional overheads, we choose to use the variable ordering given by the decision variable selection heuristic already present in the solver as the priority ordering for Delayed BCP. In particular, we use EVSIDS as the priority ordering.

Algorithm 1: Delayed BCP

```
Input: Propagation queue (priority queue) $pq$.
Output: A falsified clause if one exists.
while $pq.size() > 0$ do
    $p \leftarrow pq.pop()$;
    $assign(p)$;
    foreach clause $c$ rendered unit by $p$, resulting in
      unit literal $l$ do
        if $c$ is falsified then
            $pq.clear()$;
            return $c$;
        else if $pq.contains(-l)$ then
            $assign(-l)$;
            $pq.clear()$;
            return $c$;
        else if not $pq.contains(l)$ then
            $reason[var(l)] \leftarrow c$;
            $pq.push(l)$;
        end
    end
end
return no conflict;
```

II. Combining BCP modes through MAB

To reap the benefits of both Immediate BCP and Delayed BCP, we choose to use Reinforcement Learning (RL) methods to switch adaptively between the two BCP modes. Specifically, we represent the problem of choosing between the BCP modes as a MAB problem. We consider two different reward schemes for the RL agent, and implement them as the MapleCaDiCaL_LBD and MapleCaDiCaL_PPD solvers:

1) Literal Block Distance (LBD) [1] – this is a measure of learnt clause quality. We compute the average LBD since the last restart, and compare it to the historical average.
2) Propagations per Decision (PPD) – this is the propagation rate, computed as the number of propagations since the last restart divided by the number of decisions since the last restart.

using learnt clause quality – measured using the average Literal Block Distance (LBD) [1] – as a reward function for the RL agent. To isolate the performance of the chosen BCP strategy when computing rewards, we choose to switch

Solver	α and β decay	Score decay
MAPLECADICAL_LBD_990_275	0.990	0.275
MAPLECADICAL_LBD_990_500	0.990	0.500
MAPLECADICAL_PPD_500_500	0.500	0.500
MAPLECADICAL_PPD_950_950	0.950	0.950

TABLE I
SELECTED EXPONENTIAL DECAY HYPERPARAMETER VALUES

between BCP modes only upon solver restarts. This ensures that the structure of the implication graph relies solely on the selected BCP mode. We use a Thompson sampling approach augmented with exponential averaging as the MAB decision agent for selecting between BCP strategies.

A. Thompson Sampling

Thompson sampling [5], [6] is a stochastic approach to the MAB problem which encourages exploration at the beginning of the search, and which focuses on exploitation of the best known option as the search progresses. The method associates a beta-distributed random variable with each “arm” of the bandit, and samples a value from each distribution each time the solver restarts. The BCP variant with the largest associated sampled value is selected for the next portion of the search.

B. Exponential Moving Averages

The behaviour of BCP in a CDCL SAT solver depends heavily on the clauses present in the solver in the current state. Since the clause database evolves significantly as the solver learns and deletes clauses, it is intuitively beneficial to place a greater emphasis on the recent performance of the MAB agent than on historical performance from much earlier in the search. We address this issue in our implementation by applying exponential decays to the α and β values which parameterize the beta-distributions used by Thompson sampling, and by using an exponential moving average to represent the historical performance of the MAB agent.

These decays are applied upon every solver restart. When the score for the previous round exceeds the historical score, the trial is counted as a success, and the α value for the corresponding beta distribution is incremented by 1. Otherwise, the trial is considered to be a failure, and the β value is incremented by 1. This means that for a decay $0 < d < 1$, the maximum α and β values are bounded by $\frac{1}{1-d}$, whereas the score values are bounded by the maximum LBD.

III. IMPLEMENTATION AND SOLVER DESCRIPTIONS

The CADICAL solver [2] has a few similar implementations of Immediate BCP which are used in various places throughout the solver. Since the Priority BCP method is largely concerned with the types of clauses learnt by the solver, we only modify the implementation of BCP associated with the solver's main search loop and do not change the propagation implementations used during failed literal probing or clause vivification.

Our MAPLECADICAL solvers implement Delayed BCP alongside the existing implementation of Immediate BCP and uses RL to switch between them. We use the C++ Boost library's implementation of beta-distributions [4] to implement Thompson sampling. We submit our solver with four different configurations (see Table I).

REFERENCES

[1] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, page 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[2] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.

[3] Jonathan Chung, Sam Buss, and Vijay Ganesh. Priority-based algorithms for boolean constraint propagation. unpublished, 2023.

[4] Nikhar Agrawal et al. Boost math statistical distributions and functions 1.81.0.

[5] William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[6] William R. Thompson. On the theory of apportionment. *American Journal of Mathematics*, 57(2):450–456, 1935.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

ESA Solvers, Kissat_MAB_Binary and AMSAT in SAT Competition 2023

Shuolin Li[1], Chu-Min Li[12], Mao Luo[3], Jordi Coll[4], Mohamed Sami Cherif[1], Djamal Habet[1] and Felip Manyà[4]
[1]*Aix Marseille Univ, Université de Toulon*
CNRS, LIS, Marseille, France
shuolin.li, mohamedsami.cherif, djamal.habet@lis-lab.fr

[2]*Université de Picardie Jules Verne*
Amiens, France
chu-min.li@u-picardie.fr

[3]*School of Computer Science, Hubei University of Technology*
Wuhan, China
luomao@hbut.edu.cn

[4]*Artificial Intelligence Research Institute*
CSIC, Bellaterra, Spain
jcoll, felip@iiia.csic.es

***Abstract*—This document describes the solvers Cadical_ESA, Kissat_MAB_ESA, Kissat_Inc_ESA, Kissat_MAB_Binary and AMSAT submitted to the 2023 SAT Competition.**

Cadical_ESA, Kissat_MAB_ESA and Kissat_Inc_ESA

Based on Kissat_MAB [1], the winner of the SAT Competition 2021, Kissat_MAB_ESA tries to eliminate those unimportant variables first in inprocessing bounded variable elimination, the name ESA standing for variable Elimination Scheduled by Activity. For more details, please check our solver description in SAT Competition 2022 [2] and the paper [3]. In addition to Kissat_MAB, we also implemented ESA in Cadical [4] and Kissat_Inc [5], resulting in two new solvers Cadical_ESA and Kissat_Inc_ESA.

Kissat_MAB_Binary

In the inprocessing bounded variable elimination progress, Kissat [4] and other Kissat-based solvers only keep the resolvents of original clauses (or irredundant clauses in Kissat) containing literal x or $\neg x$ when they eliminate variable x. As for those learnt clauses (or redundant clauses in Kissat) containing x or $\neg x$, they just delete them directly, which results in a loss of information contained in those learnt clauses.

After eliminating x, clauses containing it cannot be used any more in the subsequent search, so we can't keep those learnt clauses directly, but keeping the resolvents of those learnt clauses seems a good way to preserve those learnt information. However, the number of learned clauses containing x may be several orders of magnitude larger than the number of original clauses containing x, so that keeping all the resolvents will have a huge memory cost and the benefits are far outweighed by the performance loss.

Because learnt clauses are redundant, it's not necessary for us to keep all their resolvents, but those with high quality. A clause with small lbd or size is often considered a high-quality clause. In Kissat_MAB_Binary we only keep the resolvents of binary clauses containing those eliminated variables.

AMSAT

The solver AMSAT (Amiens Marseille SAT solver) is based on lstech_maple [6], which is based on MiniSAT [7] and is the best MiniSAT based solver in SAT competition 2021. We added the following improvements, essentially based on variable activity and inspired by ESA (variable Elimination Scheduled by Activity) [3].

* *Failed literal detection, implied literal detection and equivalent literal detection.*
  If literal p is satisfied and unit propagation derives a conflict, p is a failed literal and must be falsified. If no matter the assignment of p is true or false, unit propagation satisfies a literal q, then q is an implied literal and can be satisfied. If p is satisfied, unit propagation satisfies q, and if p is falsified, unit propagation falsifies q, then p and q are equivalent literals.
  At level 0, the detected failed, implied, and equivalent literals are global and do not depend on any partial assignment. In other words, if p is failed, then $\neg p$ is satisfied directly. An implied literal q is satisfied directly. The equivalent literals are not substituted immediately, because AMSAT does not maintain a variable-to-clause structure permanently. Each equivalent literal will be substituted by an arbitrarily chosen literal in its equivalent class later upon a call to the local search solver CCAnr, together with clause subsumption to facilitate the work of CCAnr.
  Since equivalent literals are not substituted immediately, we have to deal with the non-symmetric property of unit propagation when there are clauses containing more than 2 literals. Namely, when unit propagation deduces q from p, it does not necessarily deduce $\neg p$ from $\neg q$, although $p \to q$ is equivalent to $\neg q \to \neg p$. For two equivalent literals p and q, a binary clause $\neg p \vee \neg q$ is added if unit propagation fails to deduce q from p. Note that these added binary clauses are removed when equivalent literals are substituted.

At levels other than 0, the detected failed, implied and equivalent literals depend on a partial assignment. For simplicity reason, AMSAT only detects failed and implied literals at levels other than 0, and learns an asserting clause for each failed literal and each implied literal. It is easy to derive an asserting clause for a failed literal using the usual conflict analysis. For an implied literal q, let $p \rightarrow q$ and $\neg p \rightarrow q$, a clause $C1 \vee \neg p \vee q$ ($C2 \vee p \vee q$) can be derived from the reason clause of q in the implication graph propagating p ($\neg p$), where $C1$ and $C2$ are sub-clauses contains literals falsified before p or $\neg p$ and having a path to q in the implication graph. Then, the resolvent $C1 \vee C2 \vee q$ of the two clauses asserts q.

AMSAT selects the variables in the decreasing order of their current activity (VSIDS or LRB) and detects x and $\neg x$ respectively. Let $limit$ be a parameter fixed to 100 at level 0, and 10 at other levels. the detection stops if no failed, implied or equivalent literal is found for $limit$ variables since the last failed, implied or equivalent literal is found. In order to further limit the overhead, these detections are only executed at levels 0, 1 and 2. At level 0, failed, implied and equivalent literal detection is also executed in addition before each clause vivification.

$*$ *Inprocessing bounded variable elimination*. During solving, when 2% of variables are fixed, eliminated or substituted at level 0, the inprocessing bounded variable elimination is executed. A limited part of resolvents of learnt clauses with small LBD and size containing the eliminated variables is also kept after variable elimination. All obtained resolvents from variable elimination are vivified.

$*$ *Variable mapping and space compression*. In SAT solving, if a variable becomes inactive (fixed at level 0, eliminated, etc.), we no longer need the data related to it. And if the number of inactive variables is large, there are many holes in the array used to record variable/literal data, resulting in increased traversal costs. MiniSAT-based solvers do not deal with this problem. AMSAT compresses the space by removing the holes using variable mapping. Note that the final solution generation for satisfiable instances and the unsat proof generation for unsatisfiable instances should be adapted accordingly.

$*$ *Deleting learnt clauses by variable activity*. In MiniSAT-based solvers, the indicator for deleting learnt clauses is their activity, which roughly reflects the frequencies of these learnt clauses being involved in conflicts. Let LOCAL be the subset containing clauses with LBD greater than 6 or not involved in any conflict since 30000 conflicts. The half of lower activity learnt clauses in LOCAL is removed every 15000 conflicts. AMSAT does not use learnt clause activity but variable activity to select the clauses to remove. Concretely, let $act1$ and $act2$ be the smallest and the second smallest variable activities in a learnt clause, AMSAT sort the clauses in LOCAL in the increasing of their $(act1 \times w + act2)/(LBD * LBD)$, where w is a parameter fixed to 1000. The first half is removed every 15000 conflicts.

References

[1] M. S. Cherif, D. Habet, and C. Terrioux, “Kissat MAB: Combining VSIDS and CHB through Multi-Armed Bandit,” *SAT COMPETITION*, vol. 2021, pp. 15–16, 2021.

[2] S. Li, J. Coll, C.-M. Li, M. Luo, D. Habet, and F. Manyà, “Solvers Cadical ESA and Kissat MAB ESA in 2022 SAT competition,” *SAT COMPETITION*, vol. 2022, pp. 33–34, 2022.

[3] S. Li, C.-M. Li, M. Luo, J. Coll, D. Habet, and F. Manyà, “A new variable ordering for in-processing bounded variable elimination in sat solvers.” 2023, to appear in proceedings of IJCAI’2023.

[4] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, “CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020,” in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[5] Z. Chen, X. Zhang, S. Cai, and P. Lu, “CDCL Solvers with Improved Local Search Cooperation and Pre-processing,” *SAT COMPETITION*, vol. 2022, pp. 37–38, 2022.

[6] X. Zhang, S. Cai, and Z. Chen, “Improving cdcl via local search,” *SAT COMPETITION 2021*, p. 42, 2021.

[7] N. Eén and N. Sörensson, “An extensible sat-solver,” in *Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers 6*. Springer, 2004, pp. 502–518.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Parallel by Default – MergeSat and MergeSat-Pcasso

Norbert Manthey
nmanthey@conp-solutions.com
Dresden, Germany

Abstract—**The sequential SAT solver MERGESAT is a fork of the 2018 SAT competition winner, and adds known as well as novel improvements. MERGESAT is setup to simplify merging solver contributions into one solver. By default, MERGESAT uses parallel deterministic solving with two threads and clause sharing – also in incremental mode. For stability as well as incremental solving, an interface very close to the MINISAT 2.2 interface, as well as the IPASIR interface, are supported. The solver is not tuned towards a specific application, nor a previous competition benchmark. However, the solver supports configuration of most parameters, even in incremental mode. MERGESAT-PCASSO is an unfinished solver variant that is highly based on the ideas of the 2014 solver PCASSO, and hence uses partitioning the formula when starting solving the formula, and slowly evolves to a portfolio solver.**

I. INTRODUCTION

The CDCL solver MERGESAT is based on the competition winner of 2018, MAPLE_LCM_DIST_CHRONOBT [13], and adds several known techniques, fixes, and some novel ideas around reasoning as well as parallel solving. MERGESAT uses git to combine changes, and comes with continuous integration to simplify extending the solver further.

II. DEVELOPMENT TENETS

When given a sequential compute resource, the CDCL algorithm [14] is assumed to be the most efficient way to solve SAT. To avoid duplicating implementation effort, MERGESAT is setup to easily incorporate modifications to other solvers. This setup allows to keep up with the state-of-the-art and research. Automated testing as well as extended internal checks and proof validation help to spot merge issues early.

For parallel computing resources, portfolio solvers are assumed to be limited with respect to scalability in proof generation [9]. MERGESAT's parallel variant allows to use search space partitioning in an experimental mode. Partitioning is currently handled via assumption literals, similarly to the *cube-and-conquer* [4] approach. The key difference is that MERGESAT dynamically and recursively re-partitions the search space again if compute resources become available again [6], [7], [12]. The heuristic is to keep the sequential algorithm running as long as possible on the largest possible portion of the search space. Thanks to using assumption literals, the used base-solver does not need to implement *dependency-tracking* [10], as done in PCASSO [8]. Learnt clauses can be shared across all solver instances, and unsatisfiability proofs can be generated as done in parallel portfolio solvers [5].

MERGESAT is not tuned for a specific application or benchmark. Solver additions try to stay as close to the original behavior as possible, and can be enabled by configuration. Behavior-changing modifications are automatically detected.

Most algorithms in MERGESAT can be configured. The parameter specification can be printed to a file, to be used by tools to automatically configure the solver. Furthermore, when using MERGESAT as a library, the parameters can be configured – and tuned – via environment variables.

To improve solver maintenance, the solver is implemented in a deterministic way. Algorithms are limited or switched based on step counters instead of measured run time, as the later is highly platform specific. The parallel execution is based on *barriers* similar to MANYSAT [2], to obtain a deterministic parallel solver execution. Cross-platform determinism is work in progress: MERGESAT already replaces some of the math-library functions like *exp*, to become independent of the implementation differences for different platforms.

While CDCL, as well as variable elimination [1], use resolution as the main reasoning, other simplification techniques exist that do not follow the obvious resolution pattern. *Learnt Clause Minimization* [11] is such an example. Similarly, MERGESAT implements *look-ahead* [3], which can be used to create search decisions, as well as to partition the formula. The implemented look-ahead uses double-look ahead for the second assessed polarity, as ternary clauses are collected after propagating the first polarity.

The sequential and parallel MERGESAT can emit unsatisfiability proofs in the *DRAT* format [15]. In both modes, the generation of the proof can be verified during runtime.

The sequential solver supports incremental solving with assumption literals. Incremental solving is not yet compatible with the search space partitioning, so that the parallel solver falls back to portfolio mode with sharing.

III. IMPROVEMENTS SINCE COMPETITION 2022 VERSION

The major change to MergeSat is the default setting of using two threads by default. The formula simplification has been moved from the common initialization phase towards the thread-local execution. Hence, a thread configuration that skips simplification can start search right away. For formulas with a high simplification time, this combination allows a smaller solving time.

With the jump in GLUCOSE 2.2 from the SATELITE formula simplification to the built-in simplification, the used variable elimination did not use gate-detection anymore. To close this gap, syntactic AND-gate detection has been added back to MERGESAT. Furthermore, a brief semantic search is supported to also search for gates. In contrast to KISSAT, this search is implemented directly into the main solver object.

The 2022 parallel solver used docker images that have not been based on the default images from AWS, because those images did not contain a recent-enough glibc that supports using transparent huge pages for the parallel solver. To reduce the difference between solvers, the 2023 images are based on the default images; and the scripts to setup this infrastructure are shipped as part of the MERGESAT repository. However, the used solver binary is still built in a huge-page enabled container, and hence, can still benefit from enabled huge pages.

The major difference in the parallel solver is the ability to solve with partitioning enabled. This feature is disabled in the parallel MERGESAT, but enabled in MERGESAT-PCASSO. We believe that parallel solvers scale better when using partitioning, as this way we can help diversify the search better in the search space. Similarly to portfolio solving, we can still consume all shared learned clauses, as we partition via assumption literals. We use one-sided double-look-ahead for partitioning, and attempt to create 8 partitions per node, using 3 literals each time. All solvers except the initial solver solve partitions, as well as dynamically re-partition them to create more work items – but also keeping the original partition. This way, we create a tree of partitions. In contrast to PCASSO, we use unit clauses for the partitioning constraints, partitioning can be done via assumptions – and no specialization has to be applied to the used sequential solver; drastically simplifying future updates to the sequential solver. While this does not allow us to simplify the partitions, we can instead still generate proofs, as well as share all learned clauses. From a proof-complexity point of view there is still an open question whether using only unit clauses for partitioning might result in less powerful behavior: by proving all partitions except one unsatisfiable, we can conclude that the partition clauses from the remaining partition follow from the original formulas. With PCASSO's partitioning approach, this would have allowed to also add larger clauses to the proof.

Note, from PCASSO we did not port the *only-child* approach, which un-assigns a solver in case two nodes in the tree are found to be equivalent by evaluating all siblings of a node to false. Consequently, the parallel solver will currently turn from a parallel solver into a portfolio solver, as in the worst case, on each level of the recursive partition tree exactly one child nodes is left unevaluated – in which case all nodes are equivalent and represent the same part of the search space. For n parallel solvers, this situation will be reached after creating, or cutting-off, $2^(n-1)$ search partitions. As solving hard formulas might still benefit from recursive partitioning, this is one of the outstanding features to be ported from PCASSO.

Changes to the solver are tracked in a *CHANGELOG* file. Updates to this file are enforced via automated checks.

IV. SUBMITTED SOLVERS

A. Sequential Solvers

MERGESAT is submitted in four different configurations. These configurations allow to measure the performance difference of using sequential vs parallel solving. The used sequential configuration is the "best known" sequential configuration (based on previous competition results). Furthermore, the remaining submitted configurations allow to asses the power of gate extraction during variable elimination for the current implementation. Note: no execution limits or similar mechanisms to prevent the detection to consume too much time are used in the current version.

1) Default Configuration - MERGESAT_2THREADS: This configuration uses the setup as described above, most importantly two threads are used. The second configuration does not use simplification.

2) MERGESAT_THREAD1: This configuration uses the same configuration as the default, but only uses a single thread. This thread is using the default configuration of MERGESAT; which is pretty close to the configuration used in 2021.

3) MERGESAT_BVE_GATES: This configuration is the same configuration as MERGESAT_THREAD1, except that bounded variable elimination attempts to extract AND-gates syntactically. For variable eliminations with detected gates, only the reduced resolvents are generated.ad the memory subsystem with avoidable memory accesses.

4) MERGESAT_BVE_SEMGATES: This configuration is the same configuration as MERGESAT_BVE_GATES, except that bounded variable elimination attempts to extract gates semantically before the syntactic extraction.

B. Parallel Solvers

MERGESAT participates in the parallel track. The only difference to the solver submitted to the main track is the number of used cores. The solver will analyze the number N of cores available, and then automatically chooses $\frac{N}{2}$ as the number of threads to use. The used infrastructure is based on the default infrastructure of AWS. While we set the relevant environment variable to use transparent huge pages as part of glibc, we do not enforce using the relevant glibc version to actually make use of this improvement.

MERGESAT-PCASSO also participates in the parallel track. All solvers within this solver are configured similar to MERGESAT. Furthermore, the same amount of solvers is used.

V. AVAILABILITY

The source of MERGESAT is publicly available under the MIT license at https://github.com/conp-solutions/mergesat. The version with the git tag "v4.0-rc4" is used for all MERGESAT-related submissions. The submitted starexec package can be reproduced by running "./scripts/make-starexec.sh" on this commit. The parallel variant MERGESAT-PCASSO is available at the tag "v4.0-rc4-pcasso".

The parallel variant of MERGESAT has a few open issues: tuning the default configuration, improving handling of special cases like lazily initializing and synchronizing parallel solvers during incremental solving, as well as combining incremental solving with search space partitioning and proof generation. Furthermore, tuning the different configurations of the different threads for the benchmarks of a specific application, or previous competitions, has not been done.

Acknowledgment

The author would like to thank the developers of all predecessors of MergeSAT, and all the authors who contributed the modifications that have been integrated. Furthermore, we thank the ZIH of TU Dresden for making compute resources available that have been used to develop earlier versions of MergeSAT. Adhemerval Zanella made transparent huge pages easily accessible via his modifications to glibc.

References

[1] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *SAT 2005*, ser. LNCS, F. Bacchus and T. Walsh, Eds., vol. 3569. Heidelberg: Springer, 2005, pp. 61–75.

[2] Y. Hamadi, S. Jabbour, and L. Sais, "ManySAT: a parallel SAT solver," *JSAT*, vol. 6, no. 4, pp. 245–262, 2009.

[3] M. Heule and H. van Maaren, "Look-ahead based SAT solvers," in *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. Amsterdam: IOS Press, 2009, pp. 155–184.

[4] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding cdcl sat solvers by lookaheads," in *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, ser. HVC'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 50–65. [Online]. Available: https://doi.org/10.1007/978-3-642-34188-5_8

[5] M. J. H. Heule, N. Manthey, and T. Philipp, "Validating unsatisfiability results from clause sharing parallel sat solvers," 2014, submitted.

[6] A. Hyvärinen, T. Junttila, and I. Niemelä, "A distribution method for solving SAT in grids," in *SAT 2006*, ser. LNCS, A. Biere and C. P. Gomes, Eds., vol. 4121. Springer, 2006, pp. 430–435.

[7] A. E. Hyvärinen and N. Manthey, "Designing scalable parallel SAT solvers," in *Theory and Applications of Satisfiability Testing – SAT 2012*, ser. Lecture Notes in Computer Science, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer Berlin Heidelberg, 2012, pp. 214–227. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31612-8_17

[8] A. Irfan, D. Lanti, and N. Manthey, "PCASSO – a parallel cooperative SAT solver," 2014.

[9] G. Katsirelos, A. Sabharwal, H. Samulowitz, and L. Simon, "Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers," in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*, M. desJardins and M. L. Littman, Eds. AAAI Press, 2013. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6421

[10] D. Lanti and N. Manthey, "Sharing information in parallel search with search space partitioning," in *Proceedings of the 7th International Conference on Learning and Intelligent Optimization (LION 7)*, ser. LNCS, G. Nicosia and P. Pardalos, Eds., vol. 7997, 2013.

[11] M. Luo, C.-M. Li, F. Xiao, F. Manyà, and Z. Lü, "An effective learnt clause minimization approach for cdcl sat solvers," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 703–711. [Online]. Available: https://doi.org/10.24963/ijcai.2017/98

[12] N. Manthey, "Towards next generation sequential and parallel SAT solvers," Ph.D. dissertation, TU Dresden, 2014.

[13] V. Ryvchin and A. Nadel, "Maple_LCM_Dist_ChronoBT: Featuring Chronological Backtracking," in *Proceedings of SAT Competition 2018*, 2018. [Online]. Available: http://hdl.handle.net/10138/237063

[14] J. P. M. Silva and K. A. Sakallah, "GRASP - a new search algorithm for satisfiability," in *ICCAD 1996*. Washington: IEEE Computer Society, 1996, pp. 220–227.

[15] N. Wetzler, M. Heule, and W. A. H. Jr., "Drat-trim: Efficient checking and trimming using expressive clausal proof," in *SAT*, 2014, accepted.

HKIS, UKISSATINC, PAKISINC and PAHKIS in the SAT Competition 2023

Rodrigue Konan Tchinda[1,2], Clémentin Tayou Djamegni[1]
[1]*University of Dschang, Dschang, Cameroon*
[2]*University of Bamenda, Bamenda, Cameroon*
{rodriguekonanktr, dtayou}@gmail.com

***Abstract*—This document provides a description of the sequential solvers HKIS and UKISSATINC as well as the parallel solvers PAKISINC and PAHKIS submitted to the SAT Competition 2023.**

I. HKIS AND UKISSATINC

The solvers HKIS [1] is an implementation of the PSIDS heuristic on top of KISSAT [2], [3]. The PSIDS heuristic is used for selecting polarities of branching variables. This heuristic has been integrated in numerous top-performing SAT solvers and the results of the recent SAT competitions and SAT Race (since 2019) show a remarkable increase in their performance especially in solving unsatisfiable instances. The most recent example is the 2022 SAT competition where PSIDS implemented on top of KISSAT won a gold medal in Main Sequential Track UNSAT. For the 2023 SAT Competition, HKIS is submitted to the main track with the following configurations:

.

- `psids` where the options are: `--unsat` and `--psids=true`;
- `sat_psids` where the options are `--sat`, `--phase=false` and `--psids=true`;
- `unsat` where the options are `--target=1` `--walkinitially=true` and `--chrono=true`.

During our empirical experiments, we noticed that the `--unsat` configuration applied on the original KISSAT-INC [5] performed very well on unsatisfiable instances of our testing set. We submitted this configuration to the 2023 SAT Competition and renamed the solver UKISSATINC.

II. PAKISINC AND PAHKIS

PAKISINC and PAHKIS are two parallel SAT solvers built with the PAINLESS Framework [4]. PAKISINC uses as worker one of the top-performing sequential SAT solvers of the 2022 SAT Competition namely KISSAT-INC [5]. As far as PAHKIS is concerned, it uses HKIS as worker. This year we added clauses sharing and allowed exchange of clauses with maximum LBD of 2. To participate in the Parallel Track of the 2023 SAT Competition, each of the above solvers were configured to launch 12 workers for solving formulas.

III. ACKNOWLEDGMENTS

We would like to thank the developers of PAINLESS [4], KISSAT [2], KISSAT-INC [5] and KISSAT_MAB [6].

REFERENCES

[1] R. K. Tchinda and C. T. Djamegni, “HKIS, HCAD, PAKIS and PAINLESS_ExMapleLCMDistChronoBT in the SC21,” *SAT COMPETITION 2021*, p. 26, 2021.
[2] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, “CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020,” in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.
[3] A. Biere, “CaDiCaL at the SAT Race 2019,” in *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2019-1. University of Helsinki, 2019, pp. 8–9.
[4] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, “Painless: a framework for parallel sat solving,” in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2017, pp. 233–250.
[5] X. Z. S. C. Chen, Zhihan and P. Lu., “CDCL Solvers with Improved Local Search Cooperation and Pre-processing.” *SAT COMPETITION 2022*, p. 37, 2022.
[6] M. S. Cherif, D. Habet, and C. Terrioux, “Kissat MAB: Combining VSIDS and CHB through Multi-Armed Bandit,” *SAT COMPETITION 2021*, p. 15, 2021.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

gdrcnf – solver for SAT 2023

Luke Nuttall
Independent
Yantai, China
lukerossnuttall@gmail.com

***Abstract*— This article describes the techniques used by the contest submission. Mainly clause division, developing a groebner basis and randomly guessing a solution.**

Keywords—groebner, CDCL, resolution, subsumption, garbage collection

I. Introduction

This SAT solver runs with 4 types of coroutines; Developing a groebner basis by making clause resolutions, subsuming clauses to eliminate redundancy, constructing random guesses of solutions, and garbage collection.

II. Groebner construction

A (partial) groebner basis is built for the solution set by developing resolutions of clauses as the syzygy. A 'minimal set' approach is taken to clause selection, making two distinct sets during operation; the combined and uncombined sets. The next lowest clause is taken from the uncombined set and combined with all other clauses previously combined before becoming its newest member.

The clause ordering chosen is a dynamic, entropy-based one. Lower degrees are counted first, then the average level of entropy(based on the number and length of clauses which would implicate that variable) calculated for each variable is considered, and finally a lexical ordering if the entropy is somehow equal.

III. Clause subsumption

Another coroutine will attempt to perform self-subsuming resolution on every clause. The clauses are kept in a balanced binary search tree, with a lead-degree-lexical ordering to speed up the process of testing for divisions. A companion set of benchmarks is made for this technique.

A. *Clause subsumption*

Given a problem instance with the two CNF clauses; {a,b,c & a,b} the first clause is redundant of the second clause. Any solution which satisfies the second clause implies that the first is also satisfied (it doesn't matter what c is). The second clause *subsumes* the first clause, which can be removed from the instance without changing the solution set.

B. *Self-subsuming resolution*

Given a problem instance with the two CNF clauses; {a,b,c & a,-c} neither clause subsumes the other, but the resolution of the two clauses a,b subsumes the first.

Self-subsuming resolution is the technique of replacing the clause a,b,c with a,b when it would not change the solution set (ie in light of the clause a,-c). This simplifies an applicable problem instance.

IV. Random guessing

Another coroutine will attempt to construct a solution to the problem by testing (for each variable in turn) if any clauses with that variable as lead will implicate the variable or its negation, based on the previous choices. If not, a coin is flipped. If both are implicated, a resolution of the two clauses is made and a resolution is made with every forcing clause until only guessed variables remain in a new clause to be added. This is equivalent to CDCL.

V. Garbage collection

The last coroutine will periodically delete the empty space of divided clauses and amotize sorting clauses into the orders needed by the other coroutines.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

PRS: A new parallel/distributed framework for SAT

Zhihan Chen, Xindi Zhang, Yuhang Qian, Shaowei Cai*

State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China
{chenzh,zhangxd}@ios.ac.cn,i@yuhangq.com,caisw@ios.ac.cn

I. Introduction Of Solvers Submitted to SC23

This document introduces a new lightweight parallel framework PRS, which can be seen as an improved version of ParKissat-RS [8]. PRS enters both the parallel track and cloud track, which is developed mainly based on Kissat-Inc [2], and uses many pre-processing techniques of Kissat-Pre [2]. Meanwhile, we submit a sequential SAT solver, Kissat-Pre-sc23, which is an improved version of Kissat-Pre integrating the same pre-processing techniques of PRS into Kissat-Inc [2]. The solvers submitted to SC23 are summarized in Table I.

TABLE I: Solvers Submitted to SC23

Track	Solver	Key Techniques
No-Limits	Kissat-Pre-sc23	Pre-processing before Kissat-Inc
Parallel	PRS-sc23	Pre-processing, Random Shuffle
	PRS-nopre-sc23	Random Shuffle with clause sharing
Cloud	PRS-distributed-sc23	Distributed PRS

II. Kissat-Pre-sc23

Kissat-Inc improves Kissat-MAB by employing the phase management strategy of LSTech-Maple [7] to the *target* phase. Kissat-Pre-sc23 extends the pre-processing techniques of Kissat-Inc with RC, Fourier-Motzkin Variable Elimination (FME) [3] and Probabilistic Simulation.

- Resolution Checking (RC): We use $s(l)$ denote the clauses set that literal l shows. For each variable x, if $|s(x)| \times |s(\neg x)| \leq |s(x)| + |s(\neg x)|$, we will do resolution between the two clause sets $s(x)$ and $s(\neg x)$, and checking whether all resulting clauses are always true. If so, the clauses related to variable x can be removed. The neighbor variables of success-resolved variables are set to be checked again in the next turn. RC is a variant of the Bounded Variable Elimination, with stricter conditions.
- Probabilistic Simulation: We found that the CDCL solvers perform poorly in some circuit instances, such as multiplier instances and cryptography instances. So we try to identify the circuit structure in the CNF formula and solve it through a Probabilistic Simulation algorithm [6].

This work is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDA0320000 and XDA0320300, and NSFC Grant 62122078.

III. PRS

PRS is a simple, efficient, and generic parallel SAT framework. PRS is a parallel portfolio framework that supports pre-processing, clause sharing, and many other popular parallel techniques. The diversity for each thread mainly comes from randomly shuffling (RS) [8] the initial branching order. On the base of PRS framework, we developed the following three versions for the parallel track and the cloud track.

A. PRS-sc23

PRS-sc23 is built upon Kissat-Inc, whose clause sharing strategy is the same as HordeSat [1]. PRS uses many pre-processing techniques, which can be found in [8]. RS is used in default to introduce diversity for each thread. In detail, PRS-sc23 set up a circular queue that stores all the variables in the order of their internal indices in Kissat-Inc, and the circular queue is equally divided into n blocks, $B_1, ..., B_n$. For the i-th thread, the initial variable branching order begins from B_i and ends with B_{i-1}. Note that n is the number of threads, the next block of B_n is B_1, and B_{i-1} is B_n when $i = 0$.

B. PRS-nopre-sc23

PRS-nopre-sc23 disables the pre-processing techniques of PRS-sc23.

C. PRS-distributed-sc23

PRS-distributed-sc23 is built upon PRS-sc23. Besides Kissat-Inc, PRS-distributed-sc23 integrates Maple-COMSPS [4] as another base solver. For each computer node, PRS-distributed-sc23 runs an instance of PRS-sc23 with 16 threads. There are 4 types of nodes: SAT mode group, UNSAT mode group, DEFAULT mode group, and MAPLE group. Table II shows the configurations for each group.

In the parallel version, the learnt clauses of one thread are broadcasted directly to all the other threads. In the distributed version, the clauses sharing method between the threads in the same computer node is the same as in the parallel version; but the clause-sharing method between computer nodes is quite different, the learnt clauses from node N_i can only be sent to the node N_{i+1}, where the next node of the last node is the first node.

TABLE II: Configurations for each computer node group. For each type of node group (Group), the base solver (Base Solver) and options (Option) are given, and the percent of the number of nodes in the specified mode to the total number of nodes is given in the 'Percent' column. We also give the maximum number of shared literals in the 'Limits' Column, and whether using the diversification method as pakis [5] in the 'Diversity' Column.

Group	Base Solver	Option	Limits	Diversity	Percent
SAT	Kissat-Inc	–sat	1500	Enabled	1/8
UNSAT	Kissat-Inc	–unsat	3000	Disabled	2/8
DEFAULT	Kissat-Inc	–default	1500	Enabled	1/2
MAPLE	Maple-COMSPS		3000	Disabled	1/8

References

[1] T. Balyo, P. Sanders, and C. Sinz. Hordesat: A massively parallel portfolio sat solver. In *SAT 2015*, pages 156–172, 2015.

[2] Z. Chen, X. Zhang, S. Cai, and P. Lu. Cdcl solvers with improved local search cooperation and pre-processing. *SAT COMPETITION 2022*, page 37.

[3] G. B. Dantzig and B. C. Eaves. Fourier-motzkin elimination and its dual. *J. Comb. Theory, Ser. A*, 14(3):288–297, 1973.

[4] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart. Maple-comsps, maplecomsps lrb, maplecomsps chb. *Proceedings of SAT Competition*, 2016, 2016.

[5] R. K. Tchinda and C. T. Djamegni. Hkis, hcad, pakis and painless exmaplelcmdistchronobt in the sc21. *SAT COMPETITION 2021*, page 26.

[6] S. Wu, C. Wang, and Y. Chen. Novel probabilistic combinational equivalence checking. *IEEE Trans. Very Large Scale Integr. Syst.*, 16(4):365–375, 2008.

[7] X. Zhang, S. Cai, and Z. Chen. Improving cdcl via local search. *SAT COMPETITION 2021*, page 42, 2021.

[8] X. Zhang, Z. Chen, and S. Cai. Parkissat: Random shuffle based and pre-processing extended parallel solvers with clause sharing. *SAT COMPETITION 2022*, page 51.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

DPS-Kissat

Hidetomo Nabeshima Tsubasa Fukiage Yuto Obitsu
University of Yamanashi
Yamanashi, JAPAN
nabesima@yamanashi.ac.jp

Katsumi Inoue
National Institute of Informatics
Tokyo, JAPAN
inoue@nii.ac.jp

***Abstract*—DPS is a framework for easily constructing efficient deterministic parallel SAT solvers, providing the delayed clause exchange technique introduced in ManyGlucose. We applied DPS to Kissat to construct a portfolio parallel SAT solver DPS-Kissat.**

I. Introduction

DPS is a framework for easily implementing deterministic portfolio parallel SAT solvers for shared memory multicore environment, that guarantee reproducible behavior. Reproducibility means that the execution result (the running time and a found model if satisfiable) does not change across runs. DPS is a successor to the deterministic parallel SAT solver ManyGlucose [1], from which it extracts and generalizes the mechanisms necessary to achieve reproducible behavior. We applied DPS to Kissat [2], one of the state-of-the-art sequential SAT solvers, to construct a portfolio parallel SAT solver DPS-Kissat. The version submitted to SAT Competition 2023 is essentially the same as that of SAT Competition 2022, but the PaKis [3] parameters on portfolio strategies has been disabled.

II. Delayed Clause Exchange

In parallel SAT solvers, reproducibility is lost when learnt clauses are exchanged asynchronously. Synchronous clause exchange ensures reproducibile behavior, but increases latency. The delayed clause exchange introduced in ManyGlucose allows a certain delay in the timing of clause exchanges, thereby absorbing fluctuations in the exchange interval and can reduce reducing the waiting time. However, implementing delayed clause exchange requires expert knowledge of concurrent programming, so introducing it into existing sequential SAT solvers is a time-consuming task. We have extracted the delayed clause exchange method from ManyGlucose and developed a framework DPS with a generic interface to facilitate its integration into existing sequential solvers.

DPS-Kissat is a deterministic parallel SAT solver that applies the delayed clause exchange provided by our framework to Kissat.

III. Portfolio Strategy

The diversity strategy of DPS-Kissat consists of the following three elements:

1) random variable selection until the first conflict occurs except for the first thread. The random seeds use different values for each thread.
2) disabled elimination in half of threads.

The first strategy was introduced in ManySAT 2.0 [4], the first deterministic parallel SAT solver. Clause exchange in non-deterministic parallel SAT solvers is one of the causes of search diversity due to its asynchronous nature, but this is not expected in deterministic solvers, so strategies such as random decision are necessary to ensure diversity. The second strategy was introduced because there were some instances where a lot of time was spent on in-processing.

IV. Implementation

DPS-Kissat parallelizes Kissat-SC2021 [5], which required about 400 lines of modification to Kissat and about 400 lines for the wrapper class to incorporate Kissat into DPS. The version submitted to SAT Competition 2023 launches 32 threads. The results are guaranteed to be reproducible. DPS supports non-deterministic mode, which is also entered in the competition as NPS-Kissat.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Numbers JP20H05963, JP20K11934, JP23K11214. In this research work we used the supercomputer of ACCMS, Kyoto University.

References

[1] H. Nabeshima and K. Inoue, "Reproducible efficient parallel SAT solving," in *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT 2020), LNCS 12178*, 2020, pp. 123–138.

[2] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020," http://hdl.handle.net/10138/318450, 2020, SAT Competition 2020 Solver Description.

[3] R. K. Tchinda and C. T. Djamegni, "hKis, hCaD, PaKis and PaInleSS_ExMapleLCMDistChronoBT in the SC21," http://hdl.handle.net/10138/333647, 2021, SAT Competition 2021 Solver Description.

[4] Y. Hamadi, S. Jabbour, C. Piette, and L. Sais, "Deterministic parallel DPLL," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 4, pp. 127–132, 2011.

[5] A. Biere, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba entering the SAT competition 2021," http://hdl.handle.net/10138/333647, 2021, SAT Competition 2021 Solver Description.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

New Concurrent Painless solvers based on `Kissat-MAB`: `P-KISSAT` and `P-KISSAT-STR`

Vincent Vallade*, Souheib Baarir‡*, Julien Sopena*†,
*Sorbonne Université, LIP6, CNRS, UMR 7606, Paris, France
†INRIA, Delys Team, Paris, France
‡ EPITA, France

***Abstract*—This paper describes the solvers `P-KISSAT` and `P-KISSAT-STR` submitted to the parallel track of the SAT Competition 2023.**

I. Introduction

`P-KISSAT` and `P-KISSAT-STR` are concurrent portfolio-based [1] solvers built using the Painless framework [2]. These solvers modify the sharing strategy of `parkissat-rs`, a parallel solver submitted in the 2022 SAT competition [3] and bring asynchronous strengthening of learnt clauses.

II. `P-KISSAT`

A. `parkissat-rs`

We present here what has not changed from `parkissat-rs`. It is a parallel solver based on `Kissat-MAB` [4]. We did not change the underlying implementation of `Kissat-MAB`, nor the diversification mechanism used by the parallel solver, for which we note the following:

- The solver first simplifies the formula using *Equivalent Literal Substitution* and *Resolution Checking*, methods described in [5].
- Diversify `Kissat-MAB` by giving different values for the following parameters: *stable*, *target* and *phase* and have one of the solver of the portfolio augmented by CCAnr [6], a stochastic local search algorithm, to set the phase of its variables.
- Randomly shuffle the initial branching order for each underlying solver.

In addition, `parkissat-rs` uses a shared-memory based sharing mechanism inherited from Painless. Each solver has an import buffer and an export buffer, implemented in the form of a lock-free list. Every time a solver learns a clause, it puts it in the export buffer and every time the solver restarts it can retrieve the clauses from its import buffer. A thread, called `Sharer`, is dedicated to retrieving the clauses from all export buffers and distributing them to the import buffers of all solvers. Every 0.5 seconds, the `Sharer` select 1500 literals (the sum of the size of the shared clauses) from each producer and dispatch them to consumers. The use of buffers and a dedicated thread allows to perform the sharing asynchronously and to avoid lock contention. Indeed, each buffer is accessed by only two threads: the thread that shared the clauses and the `Sharer`.

We will define more precisely the sharing policy used in our solver `P-KISSAT` in the following subsection.

B. Sharing Policy

We have added two mechanisms to `parkissat-rs`'s sharing policy. First, we added a dynamic filter for sharing clauses. In `parkissat-rs`, only the clauses with a LBD $<= 2$ are shared [7]. We use the mechanism defined in *hordesat* [8], which can dynamically raise and lower the LBD threshold depending on whether the solver has been determined to produce too little or too much.

Second, we replace a thread dedicated to solving the formula with a second `Sharer` thread. We then split the solvers into two groups, P_1 and P_2, so that the `Sharer` S_1 (or S_2) has P_1 (or P_2) as a producer and all the solvers in the portfolio as consumers. The result is that all solvers receive all export clauses as in the previous implementation, but the bandwidth is potentially increased by adding a second `Sharer` thread.

Figure 1 presents the architecture of our solver. The `SW`s for `SequentialWorker` represent the threads that execute the execute the `Sequential Engines` (here `Kissat-MAB`). We show two sharers who are responsible for distributing clauses to all solvers, but only for retrieving clauses from their producers (double-headed arrows). We define in the next section the special `Sequential Engines` named `Reducer`.

III. `P-KISSAT-STR`

`P-KISSAT-STR` has the same architecture as above, but we add two instances of a component called `Reducer`, one

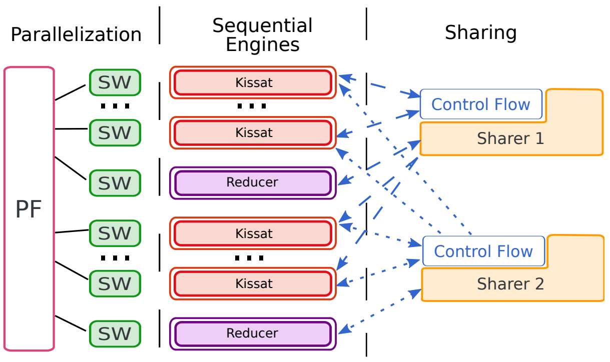


Fig. 1. Architecture of `P-KISSAT-STR`

for each group of producers. This component implements the algorithm introduced in [9] to minimize the size of the learnt clause it receives. This algorithm relies on a CDCL engine, in this case a `MapleCOMSPS` [10] solver. Adding a `Reducer` as a producer of a group allows the addition of a stream of minimized learnt clauses. We take advantage of the sharing architecture to make this minimizing of clauses asynchronous. Since a strengthened clause subsumes the original clause, it is likely that the solvers will forget the original clause over time. The `Reducer` is able to infer the empty clauses and thus give the UNSAT answer, but it cannot find a solution.

REFERENCES

[1] Y. Hamadi, S. Jabbour, and L. Sais, "Manysat: a parallel sat solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 245–262, 2009.

[2] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, "Painless: a framework for parallel sat solving," in *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 233–250, Springer, 2017.

[3] X. Zhang, Z. Chen, and S. Cai, "Parkissat: Random shuffle based and pre-processing extended parallel solvers with clause sharing," in *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*, p. 51, Department of Computer Science, University of Helsinki, Finland, 2022.

[4] M. Sami Cherif, D. Habet, and C. Terrioux, "Un bandit manchot pour combiner CHB et VSIDS," in *Actes des 16èmes Journées Francophones de Programmation par Contraintes (JFPC)*, (Nice, France), June 2021.

[5] Z. Chen, X. Zhang, S. Cai, and P. Lu, "Cdcl solvers with improved local search cooperation and pre-processing," in *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*, p. 51, Department of Computer Science, University of Helsinki, Finland, 2022.

[6] S. Cai, C. Luo, and K. Su, "Ccanr: A configuration checking based local search solver for non-random satisfiability," in *Theory and Applications of Satisfiability Testing – SAT 2015* (M. Heule and S. Weaver, eds.), (Cham), pp. 1–8, Springer International Publishing, 2015.

[7] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers.," in *IJCAI*, vol. 9, pp. 399–404, 2009.

[8] T. Balyo, P. Sanders, and C. Sinz, "Hordesat: A massively parallel portfolio sat solver," in *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 156–172, Springer, 2015.

[9] S. Wieringa and K. Heljanko, "Concurrent clause strengthening," in *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 116–132, Springer, 2013.

[10] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "Maplecomsps lrb vsids, and maplecomsps chb vsids," pp. 20–21, 2017.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

pKisDS: Dynamic Clause Sharing with Bandit Algorithms

Zhihui Xie‡, Xu Liu‡, Wanqian Luo†, Junhua Huang†, Hui-Ling Zhen†,
Xijun Li†, Mingxuan Yuan† and Shuai Li‡
†Huawei Noah's Ark Lab
‡Shanghai Jiao Tong University
{luowanqian1, huangjunhua15, zhenhuiling2, xijun.li, Yuan.Mingxuan}@huawei.com
{fffffarmer, liu_skywalker, shuaili8}@sjtu.edu.cn

***Abstract*—In this paper, we present two parallel solvers that incorporate selective clause sharing into the parallel framework. These solvers build upon the idea of selectively sharing learned clauses of the solvers to balance learning and efficiency. To explicitly model this trade-off, we utilize bandit algorithms to select which workers to share with, allowing the workers to dynamically change their sharing strategy after each restart.**

I. The Trade-off behind Clause Sharing

Modern SAT solvers heavily rely on the technique of conflict-driven clause learning (CDCL). For CDCL solvers, there are mainly two assets learned during the solving process: heuristic scores and clauses. In the context of parallel SAT solving, learned clauses have been widely used to enhance cooperation by sharing among the solvers. However, learned clauses result in a trade-off between sufficient learning and efficiency [1], [2]. Specifically, learning more lemmas is advantageous for diverse reasons, but too many clauses lead to propagation inefficiency. Besides, the roles of learned clauses differ between SAT and UNSAT instances [2]: while learned clauses are useful to accumulate to prove unsatisfiability, they tend to play insignificant roles on SAT instances.

For parallel solvers, a natural question is how to dynamically adjust the clause sharing strategy when solving an instance. This consists of two factors [3]: which clauses to share and which workers to be shared with. In this work, we consider the second problem and design an adaptive strategy that allows the receiver cores to choose their emitter cores when solving an instance. We design a bandit-based method that leverages information acquired between restarts as an indication of quality of shared clauses. Our approach bears resemblance to previous work [1], [4], which also consider controlled clause sharing.

II. Solver Submission

Our approach builds upon ParKissat-RS [5], a portfolio-based parallel solver that won first place in the 2022 SAT Competition. Each thread runs a Kissat-MAB solver [6] and shares no more than 1500 literals every 0.5s, from clauses with LBD smaller or equal to 2. The same diversification and pre-processing methods of ParKissat-RS are applied.

To dynamically adjust the sharing strategy, we formulate the problem of which workers to share with as a combinatorial bandit problem [7]. Let N be the total number of workers (or threads) and $K < N$ be the number of emitter workers from which each worker receives clauses. We fix $K = 0.75N$, suggesting that each worker receives learned clauses from 75% of the other workers. This selection process is controlled by an individual combinatorial bandit model attached to each worker. At each time the model selects a subset of K arms (i.e., emitters) to play, based on information obtained throughout the solving.

Since our desire is to select the clauses that can best accelerate solving, we need to estimate the efficiency of each worker. We consider the explored sub-tree measure [6], [8] as the reward function that estimates the efficiency of the shared clauses between restarts. Each worker at each restart updates its accompanied bandit model by assigning the same reward to all the selected arms, and then adjusts its selection accordingly.

We name our solver *pKisDS* and propose two variants for SAT Competition 2023.

A. pKisDS

The base pKisDS model uses the asymptotic optimal UCB algorithm [9] to ranks arms according to the following value:

$$\mathrm{AsymUCB}(a) = \hat{r}_t(a) + \sqrt{\frac{2\log f(t)}{T_t(a)}},$$
$$f(t) = 1 + t\log^2(t),$$

where $\hat{r}_t(a)$ is the empirical mean of the rewards received by arm a before restart t and $T_t(a)$ is the number of received rewards by arm a before restart t.

B. pKisDS-step

pKisDS-step differs from *pKisDS* slightly in that, at each restart, each worker only reschedule one of its emitters instead of all. This could empirically lead to more stable learning process and final performance [4].

References

[1] Y. Hamadi, S. Jabbour, and J. Sais, "Control-based clause sharing in parallel sat solving," *Autonomous Search*, pp. 245–267, 2012.
[2] C. Oh, "Between sat and unsat: the fundamental difference in cdcl sat," in *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18*. Springer, 2015, pp. 307–323.

[3] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, “Painless: a framework for parallel sat solving,” in *Theory and Applications of Satisfiability Testing–SAT 2017: 20th International Conference, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings 20*. Springer, 2017, pp. 233–250.

[4] N. Lazaar, Y. Hamadi, S. Jabbour, and M. Sebag, “Cooperation control in parallel sat solving: a multi-armed bandit approach,” Ph.D. dissertation, INRIA, 2012.

[5] X. Zhang, Z. Chen, and S. Cai, “Parkissat: Random shuffle based and pre-processing extended parallel solvers with clause sharing,” *SAT COMPETITION 2022*, p. 51.

[6] M. S. Cherif, D. Habet, and C. Terrioux, “Combining vsids and chb using restarts in sat,” in *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[7] W. Chen, Y. Wang, and Y. Yuan, “Combinatorial multi-armed bandit: General framework and applications,” in *International conference on machine learning*. PMLR, 2013, pp. 151–159.

[8] A. Paparrizou and H. Wattez, “Perturbing branching heuristics in constraint solving,” in *Principles and Practice of Constraint Programming: 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7–11, 2020, Proceedings 26*. Springer, 2020, pp. 496–513.

[9] T. Lattimore and C. Szepesvári, *Bandit algorithms*. Cambridge University Press, 2020.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Mallob$\{32,64,1600\}$ in the SAT Competition 2023

Dominik Schreiber
Institute of Theoretical Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany
dominik.schreiber@kit.edu

***Abstract*—We describe our submissions of *Mallob* to the parallel and cloud tracks of the SAT Competition 2023. Our changes mostly aim at reducing (computational and memory) overhead and at reducing turnaround times of shared clauses in order to reduce redundant work performed.**

***Index Terms*—Parallel SAT solving, distributed SAT solving**

I. Introduction

In this report we describe the configurations of our scheduling and SAT solving system **Mallob** [1], [2] which we submit to this year's International SAT Competition. As in prior years [3]–[5] we configure our system to immediately schedule a single instance (i.e., the problem input) with full demand of resources and to quit after its processing. While this mode of operation now supports producing UNSAT proofs [6], we are not using this functionality since it adds overhead and limits the set of usable solvers.

II. Overview and Setup

In contrast to previous submissions [3]–[5] where we spawn one MPI process for each group of four hardware threads — a heritage from Mallob's precursor HordeSat [7] — we now spawn only one MPI process for each physical machine. This change reduces overhead in terms of run time and memory usage.[1] It also allows running Mallob as a shared-memory parallel solver without MPI on a single large machine. However, the increased degree of concurrency within each process also uncovered issues in some of our concurrent data structures that were previously "good enough" when only using four threads. For this reason we rewrote large portions of Mallob's data structures for handling produced clauses (see Section III-C). As last year [5], we run solver threads within a separate sub-process that is forked from the respective MPI process. Since restarting a solver process that orchestrates dozens of solvers leads to significant loss of progress, we also adjusted our *memory panic* mechanism [5] to gracefully terminate and clean up individual solvers in a solver process. For cases where an out-of-memory situation occurs despite our precautions, each subprocess now adjusts its out-of-memory score (`oom_score_adj`) in such a way that it is the first process to be killed by the operating system. Killing a SAT subprocess is always preferrable to killing an MPI process, since the latter crashes the distributed program.

[1]In particular, each MPI process keeps a copy of the problem input and additionally writes a copy to a shared-memory segment.

We submit two parallel versions and one distributed ("cloud") version of Mallob. We employ 32 (64) Kissat instances[2] in the parallel configuration **Mallob32** (**Mallob64**) and employ a mix of 800 Kissats, 533 CaDiCaLs, and 267 Lingelings in the distributed configuration **Mallob1600**. Compared to last year we omit Glucose, which might be slightly detrimental to overall performance but simplifies our setup and also renders all parts of our submission Free Software.

Inspired by recent work that involved modifications to CaDiCaL [6], we enhanced the clause export implementation of our Kissat backend: Instead of exporting clauses at conflict analysis, we now export any new redundant clause that is created (and any unit that is fixed). As such, solvers can now also share insights gained from inprocessing techniques such as probing, vivification, or hyper-ternary resolution. In addition, we now allow Kissat to import incoming clauses whenever at decision level zero without waiting for a certain number of conflicts to occur in between (500 conflicts in our previous submissions), which can reduce turnaround times of shared clauses (see Section III-B). We also added some minor improvements to Kissat's clause import code and extended its portfolio to a total of 15 distinct configurations.

III. Clause Sharing

Regarding Mallob's clause sharing, we introduce a change in handling LBD scores; an increase of the frequency at which all-to-all clause sharing is performed; and improvements to Mallob's clause filtering and buffering data structures.

A. Handling LBD Scores

We have integrated a technique that was already featured in TopoSAT 2 [8]: If a solver imports a clause, the clause's LBD value is reset to the clause's length, contrary to our (HordeSat's) earlier approach of importing each clause with its original LBD. The TopoSAT 2 approach takes into account that LBD is a local metric that depends on the solver state and therefore may not be meaningful for all solvers globally. Note that the HordeSat approach may force solvers to keep an unsustainable number of low-LBD clauses over time while the TopoSAT 2 approach rather results in solvers discarding many incoming clauses after a few conflicts.

[2]In last year's competition, a misconfiguration lead to our submission "Mallob-Ki" to use Lingeling instead of Kissat as a solver backend. See http://algo2.iti.kit.edu/schreiber/downloads/mallob-ki-mallob-li.pdf

B. Sharing Frequency

Since clause sharing may be considered a kind of distributed pruning of search space, we suspect that it is beneficial to minimize the latency between a clause's production by solver S and its import by a solver S'. Intuitively, lowering this "turnaround time" of a clause c may reduce the chance that S' enters a sub-space which S already reported as pruned via c. Therefore, more frequent clause sharing may decrease the amount of redundant work performed. We increased the frequency at which all-to-all clause sharing is performed from 1/s to 0.33/s. Accordingly, we scaled down the buffer limit for each sharing by a factor of three.

C. Clause Filtering and Buffering

We added some improvements and bugfixes to Mallob's exact distributed clause filtering mechanism [5]. For instance, we now use the clause metadata in hash table H to keep track of the last epoch where a local solver produced a clause and successfully wrote it into clause buffer B, and we added a periodic garbage collection which erases clauses from H whose last sharing and production both range back beyond the user-defined resharing interval. We also use this additional information to more reliably block clauses from being imported by a solver which recently produced them.

In our 2022 implementation of adaptive clause buffers B [5], each *slot* l for clauses of length l is guarded by a mutex. Inserting a clause c of length l in B requires locking slot l as well as potentially all slots $l' > l$ in succession in order to erase "worse" clauses and then use the freed budget to insert c to l. This is acceptable with just four solver threads but may not scale to our new setup. Rather than actually erasing clauses from a worse slot l', we now just *mark* a deletion by manipulating an atomic clause counter of slot l', hence we only need to lock slot l. The actual deletion takes place the next time a lock for slot l' is held. If B is close to full before flushing, then we also determine the minimum $\hat{l}$ such that $\geq 95\%$ of all clauses in B had length $\hat{l}$ or below. If a produced clause c is larger than $\hat{l}$, then it is highly unlikely that c is ever exported from B before it is deleted[3] and we discard c without attempting its insertion.

Solvers may occasionally produce large bursts (hundreds of thousands) of unit clauses, which overburdens B and results in discarding most produced clauses. For this reason, we now allow the buffers to store an unlimited number of unit clauses while keeping the shared budget for all other slots.

Lastly, we have noticed a shortcoming in the merge of clause buffers during our distributed aggregation [1]. If the set of available clauses exceeds the current aggregated buffer limit, then the buffer is truncated, returning excess clauses to the local solver process. Since clauses are sorted alphanumerically, this may introduce a slight bias to our sharing. We now randomly select the clauses from the buffer's "worst" bucket which make the cut and return the remaining clauses.

[3]The budget of B is set to $10\times$ the export limit per flush.

IV. Input Permutation

Permuting the input before handing it to a solver can be used as an additional source of diversification. We experimented with this kind of diversification in 2021 [4] but did not include it in 2022 since its implementation incurred too much overhead to be worthwhile. The formula is present as a chunk of shared memory that is parsed by many solvers concurrently, so direct manipulations of the formula should be avoided.

This year we reintroduce input permutation for all but the first ten solvers. In our new implementation, we select up to $k = 128$ input clauses to which we store a pointer. The first clause in the input is always selected while the remaining $k-1$ clauses are selected at random. Each of the k pointers represents a chunk of the input beginning at the referenced clause. These k pointers are then permuted and the input chunks are read in the corresponding order. This procedure is cache-friendly and features a non-zero probability for any pair of clauses (c_1, c_2) to be read in reverse order. In addition, the order of literals in each clause is shuffled using a single clause buffer for each solver thread.

Acknowledgment

The author thanks Armin Biere for providing the solvers Kissat, CaDiCaL, and Lingeling which our solving system is built upon. The author gratefully acknowledges the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (www.lrz.de). Some preparation for this work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).

References

[1] D. Schreiber and P. Sanders, "Scalable SAT solving in the cloud," in *Proc. SAT*, pp. 518–534, Springer, 2021.
[2] P. Sanders and D. Schreiber, "Decentralized online scheduling of malleable NP-hard jobs," in *Proc. Euro-Par*, pp. 119–135, Springer, 2022.
[3] D. Schreiber, "Engineering HordeSat towards malleability: mallob-mono in the SAT 2020 cloud track," in *Proc. of SAT Competition*, pp. 45–46, 2020.
[4] D. Schreiber, "Mallob in the SAT competition 2021," in *Proc. of SAT Competition 2021*, p. 38.
[5] D. Schreiber, "Mallob in the SAT competition 2022," in *Proc. of SAT Competition 2022*, pp. 46–47.
[6] D. Michaelson, D. Schreiber, M. J. Heule, B. Kiesl-Reiter, and M. W. Whalen, "Unsatisfiability proofs for distributed clause-sharing SAT solvers," in *Proc. TACAS*, pp. 348–366, Springer, 2023.
[7] T. Balyo, P. Sanders, and C. Sinz, "Hordesat: A massively parallel portfolio SAT solver," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 156–172, Springer, 2015.
[8] T. Ehlers and D. Nowotka, "Tuning parallel SAT solvers," *Proceedings of Pragmatics of SAT*, vol. 59, pp. 127–143, 2019.

BENCHMARK DESCRIPTIONS

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Benchmark Compilation for SAT Competition 2023

Markus Iser
HIIT, Department of Computer Science, University of Helsinki
Helsinki, Finland
markus.iser@helsinki.fi

***Abstract*—The compilation of a new set of benchmarks is one of the most important tasks in the run-up to the SAT competition. In this document, we provide detailed information about the process and the criteria that were important in compiling the benchmark set for the 2023 SAT competition.**

***Index Terms*—benchmark compilation**

I. Introduction

The compilation of benchmarks is an essential part of the SAT competition. The benchmarks for the 2023 SAT competition have been compiled to provide a fair assessment of the state of the art in SAT solving.

The "Bring your own benchmarks" (BYOB) rule reinforces the community-oriented nature of the SAT competition by allowing participants to submit their own instances. This also leads to a larger share of novel benchmark instances and a greater diversity of instance domains, which is desirable.

Thanks to the strong participation, we were able to draw a sample from a large set of new benchmarks. From the subset of new instances that were not found to be too simple or isomorphic to each other, we drew a random sample stratified by author. We also included a random set of benchmarks from the Anniversary Track of the 2022 SAT competition to balance the determined numbers of satisfiable and unsatisfiable instances in the benchmark.

II. Benchmark Compilation

In this competition, 19 authors submitted a total of 654 benchmark instance files, among which we identified 636 as unique benchmark instances. We further filtered this set of instances as described below to ensure instance hardness and diversity.

To ensure hardness of the instances, we ran Minisat with a time limit of one minute on a laptop with an AMD Ryzen 7 PRO 3700U CPU and 16 GB RAM and filtered out 39 instances that could be solved by Minisat within the time limit.

To ensure diversity, we filtered out isomorphic instances. To this end, we projected the set of instances onto an equivalence class identifier formed from the hash sum of the sorted degree sequence of the weighted literal incidence graph.

This filtering gave us a set of 527 instances that were neither too simple nor isomorphic to each other. From this set, we drew 306 instances by randomly selecting a maximum of 20 instances from each author. Table I contains detailed information about the instance domains and the numbers of submitted, filtered and selected instances for each author.

TABLE I
Benchmark selection by corresponding author.

Submitter	Domain	Unique	Filtered	Selected
Chowdhury	Production	20	20	20
Chung	UNSATcoin	20	15	15
Fleury	GRS, RISC-V	94	94	20
Gao	Interval Matching	23	23	20
Green	Register Allocation	20	20	20
Hiller	Software Verification	6	4	4
Karia	Brent Equations	20	19	19
Manthey	Cryptography	26	26	20
Mayer-Eichberger	Social Golfer	26	21	20
Niskanen	Argumentation	138	136	20
Nuttall	Subsumption Test	19	5	5
Osama	Hashtable Safety	21	21	20
Reeves	Mutilated Chessboard, PHP	20	18	18
Shuolin Li	REG^N Formulas	32	5	5
Tchinda	Scheduling	20	19	19
Xindi	Miter	20	4	4
Yldirimoglu	RPHP, XOR, Tseitin, Coloring	40	40	20
Zhang	Cryptography	51	17	17
Zheng	Set Covering	20	20	20
Total		636	527	306

TABLE II
A priori known SAT / UNSAT balance.

	SAT	UNSAT	UNKNOWN	Sum
New Benchmarks	82	70	154	306
Old Benchmarks	41	53	0	94
Total	123	123	154	400

Finally, we randomly sampled a set of 94 benchmarks from the Anniversary track of the 2022 SAT competition to balance the numbers of known satisfiable and unsatisfiable instances in the benchmark. Table II shows detailed information about the a priori known numbers of satisfiable, unsatisfiable and unknown instances in the benchmark. The instances are available for download at https://benchmark-database.de?track=main_2023.

III. Reproducibility

The script and dataset which we used for compiling the set of benchmark instances can be downloaded from the SAT competition website https://satcompetition.github.io/2023/. To run the script, the package https://pypi.org/project/gbd-tools/ must be installed [1].

References

[1] M. Iser, C. Sinz, "A Problem Meta-Data Library for Research in SAT." Proceedings of Pragmatics of SAT 2018, Oxford, UK, July 7, 2018

The Profitable Robust Production Problem

Md Solimul Chowdhury
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA.
mdsolimc@cs.cmu.edu

***Abstract*—This document describes a crafted SAT benchmark that provides a solution for a production plan for an imaginary business organization. To meet increasing demands, the business plan must ensure a non-decreasing number of products over a span of $D \geq 1$ days. The business plan must ensure that the cost of production each day must be lower than the expected net sales on that day- thus making the business potentially a profitable one.**

We call this problem Profitable Robust Production (PRP) Problem. We formulate PRP as a SAT problem. We have submitted 20 instances of PRP to the SAT Competition-2023.

I. The PRP Problem

With the past success of solving the Graceful Production Problem [2], the manager at the *FantasyElectronics* corporation is now challenged with making the business of producing the *iImagine* product with the following requirements: (i) the business must be *profitable*: each day, the expected net sales of *iImagines* must exceed the cost for producing them, (ii) the production must be (a) *steady*: the total number of product produced in a given day must not be lower than the number of produced in the previous day, and (b) *robust:* the minimum number of product produced (over all its production units) in a given day must be higher than the number of produced in the previous day. First, the manager needs to make sure a production plan exists, which respects these requirements.

The company has one production *factory* with $u \geq 1$ production units.

1 Each unit produces 0 or more gadgets each day, some of which may have manufacturing defects and are not functioning.
2 The capacity of each unit limits a maximum of $m \geq 0$ *functioning* gadgets each day.

Let e_{ij}, c_{ij}, and s_{ij} be the number product produced at unit i at day j, cost of producing a product at unit i at day j, and expected sale price of each product produced from unit i at day j, where, $1 \leq i \leq u$ and $c_{\texttt{min}} \leq c_{ij} \leq c_{\texttt{max}}$, and $s_{\texttt{min}} \leq s_{ij} \leq s_{\texttt{max}}$, and $1 \leq j \leq D$. The production venture of *iImagine* needs to fulfill the following requirements:

- `profit` At day j, $tc_j = \sum_i e_{ij} * c_{ij}$ the total cost of running unit i must be lower than $tp_j = \sum_i e_{ij} * s_{ij}$, the expected profit for that day.
- `steady` The total number of functioning *iImagaine* produced in a day needs to be higher or equal to its previous day's total, and
- `robust` For any given day, for these u units, the minimum number of gadgets produced must be greater than the minimum number of products produced in the previous day.

Is it possible for these units to run for $D \geq 2$ days, while satisfying these requirements? The manager needs to find answer of this new question.

II. SAT encoding of the PRP problem

A. PRP *as a SAT Benchmark*

Here, we encode the PRP problem as a SAT benchmark. Given a PRP problem, we encode it as a SAT formula $F_{\texttt{prp}}$ as follows

$$F_{\texttt{prp}} = F_{\texttt{profit}} \cup F_{\texttt{steady}} \cup F_{\texttt{robust}} \cup F_{\texttt{ends}}$$

, where, $F_{\texttt{profit}}, F_{\texttt{steady}}, F_{\texttt{robust}}$, and $F_{\texttt{ends}}$ are defined as follows:

$$F_{\texttt{proft}} : \bigwedge_{j=1}^{D} \sum_{i=1}^{u} e_{i,j} * c_{i,j} < \sum_{j=1}^{u} e_{i,j} * s_{i,j}$$

$$F_{\texttt{steady}} : \bigwedge_{j=1}^{D-1} \sum_{i=1}^{u} e_{i,j+1} > \sum_{i=1}^{u} e_{i,j}$$

$$F_{\texttt{robust}} : \bigwedge_{j=1}^{D-1} \texttt{min}\ (e_{1,j} \dots e_{u,j}) < \texttt{min}\ (e_{1,j+1} \dots e_{u,j+1})$$

$$F_{\texttt{ends}} : \bigwedge_{d=1}^{D} \neg e_{0,j} \wedge \neg e_{n+1,j}$$

Over D days,

- $F_{\texttt{profit}}$ encodes the profit constraint.
- $F_{\texttt{steady}}$ encodes the `steady` constraint.
- $F_{\texttt{robust}}$ encodes the robust constraint.
- $F_{\texttt{ends}}$ encodes the assertion that left unit (resp. right unit) of the leftmost (resp. rightmost) always produces 0 gadgets, which marks the horizon of the factory.

$F_{\texttt{PRP}}$ is SATISFIABLE, if the factory can run for D days by conforming to the `profit`, `steady`, `robust`, and `ends` constraints, otherwise, it is UNSATISFIABLE.

III. Problem Modeling and Instance Generation for the PRP benchmarks

A. Problem Modeling

picat [1] is a CSP solver, which accepts a CSP problem and converts it to a SAT CNF formula, which is inturn solved by a SAT solver hosted by **picat**. Before solving the converted CNF formula, **picat** outputs the CNF formula.

To generate instances for the PRP benchmark, we use this CNF generation feature of **picat**. First, we modelled the PRP problem in a **picat** program $picat_{\texttt{PRP}}$. Then, for a given set of parameter values for $(D, u, [c_{\texttt{min}}, c_{\texttt{max}}], [s_{\texttt{min}}, s_{\texttt{max}}])$ for a PRP problem, we use this $picat_{\texttt{PRP}}$ model to generate CNF $F_{\texttt{PRP}}$ by exploiting the CNF generation functionality of **picat**.

B. Instance Generation

We have generated a set of $F_{\texttt{PRP}}$ instances with the $picat_{\texttt{PRP}}$ by varying the parameters D and u, while setting m (maximum production limit of an unit per day) to a fixed value of 1,000, $c_{\texttt{min}}$ and $s_{\texttt{min}}$ to 0, $c_{\texttt{max}}$ and $s_{\texttt{max}}$ to 100. From this set of instances, we have submitted 20 instances for SAT competition-2022 (CNF file has the the following format PRP_D_u).

References

[1] Picat, http://picat-lang.org/resources.html, Accessed: 2020-04-09

[2] Md Solimul Chowdhury. The Graceful Production Problem. Proceedings of SAT Competition-2022:61-62

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

UNSATcoin

Jonathan Chung
University of Waterloo
Waterloo, Canada
ORCID: 0000-0001-5378-1136

Sam Buss
UC San Diego
La Jolla, United States of America
ORCID: 0000-0003-3837-334X

Vijay Ganesh
University of Waterloo
Waterloo, Canada
ORCID: 0000-0002-6029-2047

***Abstract*—This benchmark suite consists of unsatisfiable variants of Bitcoin mining problems. The satisfiable variants of this problem were submitted as the SATcoin benchmark for the SAT competition in 2018 by [4]. We use their problem encoding and instance generator to generate the instances for this benchmark.**

I. Introduction

We refer the reader to the original SATcoin benchmark description document [4] for an overview of the problem and the encoding. The code for generating the instances is available at https://github.com/jheusser/satcoin, and a more detailed explanation of the encoding is presented in their main document [3].

As a brief introduction, the Bitcoin mining problem searches for a nonce value such that the hash for an input message is smaller than some target value. This search is encoded as a computer program, which a bounded model checker then encodes into a SAT problem. When the input message is known to have a single nonce value satisfying the target value, the problem can be made definitely SAT or UNSAT by restricting the search space for the nonce. This is done by setting minimum and maximum values for the nonce and setting them to contain or exclude the known "correct" nonce value. The difficulty of the problem can be scaled by changing the size of the search space (i.e., increasing the distance between the minimum and maximum bounds on the nonce) [4].

The problems in this benchmark suite are parameterized by a value k, where (for a known nonce value v) the minimum and maximum bounds are set to $v+1$ and $v+2k$ respectively.

II. Instance Selection

The SATcoin instances submitted to SAT Competition 2018 are now quickly solvable by strong SAT solvers like CaDiCaL [2]. These instances used very small values of k (from 3 to 10), as well as powers of 2 from 4 up to 8192.

To select more interesting and difficult instances for this competition, we generated instances ranging k from 5000 to 20000, with a step size of 20 for $k \in [5000, 10000]$ and a step size of 100 for $k \in [10000, 20000]$.

We ran the CaDiCaL SAT solver on Intel E5-2683 v4 Broadwell 2.1 GHz CPUs [5] with 10 GB of RAM and a time limit of 5000 seconds to identify difficult problems, and selected a sample of instances from each range of k. To ensure that our sample of problems is "interesting" as defined by the competition guidelines [1], and to ensure a variety of problem difficulties, we selected a mix containing some of the most difficult instances (unsolved by CaDiCaL) as well as some easier – but not trivial – instances (solved by CaDiCaL).

The k values of the selected instances and the associated solving times for CaDiCaL are presented in Table I.

k	Solving Time (s)
5920	timeout
6120	timeout
7200	timeout
7720	1018.8
8120	timeout
9080	timeout
9880	timeout
10600	timeout
11100	1431.87
11400	timeout

(a)

k	Solving Time (s)
11900	1738.95
12300	timeout
17400	timeout
17800	2158.33
18200	timeout
18400	2418.61
18600	timeout
18800	timeout
19500	2400.43
19800	3826.57

(b)

TABLE I: CaDiCaL solving times for selected instances.

References

[1] SAT Competition 2023. SAT Competition 2023 - Benchmarks. https://satcompetition.github.io/2023/benchmarks.html, 2023.

[2] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.

[3] Jonathan Heusser. Sat solving - an alternative to brute force bitcoin mining. https://jheusser.github.io/2013/02/03/satcoin.html, 2013.

[4] Norbert Manthey and Jonathan Heusser. Satcoin – bitcoin mining via sat. In Marijn Heule, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2018: Solver and Benchmark Descriptions*, volume B-2018-1 of *Department of Computer Science Series of Publications B*, page 67. University of Helsinki, 2018.

[5] Digital Research Alliance of Canada. Graham - CC doc. https://docs.alliancecan.ca/wiki/Graham.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Verifying Floating-Point Commutativity with GRS

Robin Trüby
University Freiburg
Freiburg, Germany
robin.trueby@web.de

Mathias Fleury
University Freiburg
Freiburg, Germany
fleury@cs.uni-freiburg.de

Armin Biere
University Freiburg
Freiburg, Germany
biere@cs.uni-freiburg.de

I. GRS Description

GRS are a technique to operate on floating operations efficiently following the IEEE 754 norm [1]. It is used inside CPUs for efficiency. It is possible to prove that only 3 additional digits, called guard (G), round (R), and sticky (S). This is sufficient to keep the results correct. This is more efficient for double than converting the numbers to 80-bit floating points and then rounding back. In his Bachelor project, the first author created a webpage [2] in German to illustrate the use of GRS bits.

GRS must be paired with a rounding technique for ties. We use round to even, which is better to keep precision than always rounding down or always rounding up.

The benchmark we are interested correspond to proving that `a+b=b+a`. We know that such encoding for integer addition is very easy to solve for SAT solvers, while becoming hard for multipliers. To simplify the encoding, we first generated word-level SMT benchmarks. We have submitted some of the (smaller) benchmarks to the SMT Competition directly.

To validate the correctness of our encoding [3], we compared the results of our encoding to the floating point theory of SMT solvers and found no difference.

The idea of addition is that our two numbers a and b are binary encoding from:

$$\begin{aligned} a &= (-1)^{s_a}(1+m_a)2^{n_a} \\ b &= (-1)^{s_b}(1+m_b)2^{n_b} \end{aligned}$$

Then we align the exponents

$$\begin{aligned} a+b = &\Big((-1)^{s_a}(1+m_a)2^{n_a-\max(n_a,n_b)}+ \\ = &\quad (-1)^{s_b}(1+m_b)2^{n_b-\max(n_a,n_b)}\Big)2^{\max(n_a,n_b)} \end{aligned}$$

Finally, we have to implement the operations. For this, we used a word-level representation in our SMT encoding. Finally, we have handle various special cases, including realigning numbers, changing the exponents, handling denormal numbers, and handling overflows (like NaN or infinity). Finally we can produce the final binary result.

Due to the encoding, we know that all problems are UNSAT. Also we have not found a simple way to generate satisfiable hard benchmarks with GRS.

II. Benchmark Generation

To generate the benchmarks we have written a custom C program (available online [4]) that writes the SMT file. Then we use Bitwuzla [5] to convert the SMT files to DIMACS files without any specific options.

During experimentation, we found out that encoding as `(if a then x=s else x=t)` makes the problem very easy for SMT solver and we generate benchmarks using `x = (if a then s else t)`.

The naming convention for our benchmarks is `grs-<mantissa-size>-<exponent-size>.cnf`.

References

[1] IEEE Committee, "IEEE standard for floating-point arithmetic," IEEE Computer Society, New York, NY, USA, Standard IEEE Std 754-2008, Aug. 2008. [Online]. Available: https://web.archive.org/web/20160806053349/http://www.csee.umbc.edu/~tsimo1/CMSC455/IEEE-754-2008.pdf

[2] R. Trüby. (2023) Floating-point arithmetic. Bachelor Project at University Freiburg, in German. [Online]. Available: https://cca.informatik.uni-freiburg.de/teaching/grs-bits/home.html

[3] ——, "Generating word-level floating-point benchmarks," 2023, bachelor Thesis at University Freiburg, Submitted.

[4] ——. (2023) Bachelorarbeit. Accessed April 2023. [Online]. Available: https://github.com/Robin060500/Bachelorarbeit

[5] A. Niemetz and M. Preiner, "Bitwuzla at the SMT-COMP 2020," *CoRR*, vol. abs/2006.01621, 2020. [Online]. Available: https://arxiv.org/abs/2006.01621

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Replacing RISC-V Instructions by Others

Sonja Gurtner
Johannes-Kepler-University Linz
Linz, Austria
sonja.gurtner@gmail.com

Lucas Klemmer
Johannes-Kepler-University Linz
Linz, Austria
lucas.klemmer@jku.at

Mathias Fleury
University Freiburg
Freiburg, Germany
fleury@cs.uni-freiburg.de

Daniel Große
Johannes-Kepler-University Linz
Linz, Austria
daniel.grosse@jku.at

The RISC-V ISA is becoming increasingly popular in industry and teaching. To realize the ultimate goal of having a really “reduced” instruction set, it is possible to remove even more instructions than the ones remaining in the default RISC-V integer architecture (compared to CISC).

We are here interested in two different variants of instruction removal:

a) Golcrest-VP [1], [2].: The idea is that we consider our CPU with an internal sub-CPU that is able to execute only a single instruction, SUBLEQ (subtract and branch if less than or equal). The subsystem is properly initialized (with constants like 1, 4, or 32) when called by the CPU and has extra registers, that do not need to be restored.

For example, the addition of a and b into c would be:

```
SUBLEQ r, ZERO, import_b;     r := -b
SUBLEQ export_register, import_a, r
  ;  dest  := a - r
```

This very simple example does not feature any jump, but it is needed when doing bitwise operations.

b) Subtraction-RV [3].: Here we use subtraction and branching as two separate instructions from the CPU and results areend. The verification comes in two flavors: verification of one instruction and then assuming that all other occurrences of this instruction are correct (and therefore using the simpler one, e.g. directly use an addition instead of two subtractions) and the *nested* version, where the replacement program uses only subtraction and jumping instructions.

For example, addition of a and b would be:

```
<save the registers a, b, and c>
sub b, zero, b ; b := -b
sub c, a, b    ; c := a - b
<save the register c>
<restore all registers>
<finally restore c>
```

In the second version, it is important to be careful that operations like `add a, a, a` are correctly implemented including when `a` is a temporary register that is used to store value (refer to the Master thesis [3] for more details).

Both works were verified using the solver-aided framework Rosette [4], which translates constraints from a Racket-like language to SMT-LIB and automatically calls a SMT solver. It already does some bitblasting in order to share terms, but the resulting SMT file still contains (some) words.

In order to generate the benchmarks we used Rosette to generate SMT files and converted them to SAT files using Bitwuzla [5]. Remark that some of the smaller files have already been submitted to the SMT competition, but due to the low timeout there are few overlaps. The naming convention of the files is `oisc-goldcrest-<bitsize>.cnf` or `oisc-subrv-<operator>-[-nested]-<bitsize>-[-pc].cnf`. We have added one source of counterexamples, not excluding the program counter from the source or destination registers. Problems with `-pc` are satisfiable (the pc is increased during the replacement, hence changing its value when reading it), the others are unsatisfiable.

In terms of solving we know that `oisc-subrv-and-nested-32.cnf` is solvable by Kissat in 10 hours (with some non-default options). Therefore, we included problems only up to bit size 28. For `oisc-goldcrest-xor-16.cnf`, Kissat should be able to solve them in slightly more than 5000 s on a recent computer. However, those benchmarks seem to be easier for CaDiCaL than Kissat.

References

[1] L. Klemmer and D. Große, “An exploration platform for microcoded RISC-V cores leveraging the one instruction set computer principle,” in *IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2022, Nicosia, Cyprus, July 4-6, 2022*. IEEE, 2022, pp. 38–43. [Online]. Available: https://doi.org/10.1109/ISVLSI54635.2022.00020

[2] L. Klemmer, S. Gurtner, and D. Große, “Formal verification of SUBLEQ microcode implementing the RV32I ISA,” in *Forum on Specification & Design Languages, FDL 2022, Linz, Austria, September 14-16, 2022*. IEEE, 2022, pp. 1–8. [Online]. Available: https://doi.org/10.1109/FDL56239.2022.9925662

[3] S. Gurtner, “A formally verified reduction of the RV32I ISA,” 2022, master Thesis at Johannes-Kepler-University, Linz, Austria.

[4] E. Torlak and R. Bodík, “Growing solver-aided languages with rosette,” in *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH ’13, Indianapolis, IN, USA, October 26-31, 2013*, A. L. Hosking, P. T. Eugster, and R. Hirschfeld, Eds. ACM, 2013, pp. 135–152. [Online]. Available: https://doi.org/10.1145/2509578.2509586

[5] A. Niemetz and M. Preiner, “Bitwuzla at the SMT-COMP 2020,” *CoRR*, vol. abs/2006.01621, 2020. [Online]. Available: https://arxiv.org/abs/2006.01621

Matching of Properly Intersecting Intervals

Yu Gao
Theory Lab, Central Research Institute, 2012 Labs, Huawei Technologies Co., Ltd.
Beijing, China
gaoyu99@huawei.com

***Abstract*—This document introduces our instances for the SAT competition 2023.**

I. Description of Benchmark

The set of benchmarks is from problem K of the 2022 ICPC (International Collegiate Programming Contest) Asia East Continent Final [1].

You are given a sequence $a_0, \dots, a_{2n}$. Initially, all numbers are zero.

There are n operations. The i-th operation is represented by two integers l_i, r_i $(1 \le l_i < r_i \le 2n, 1 \le i \le n)$, which assigns i to $a_{l_i}, \dots, a_{r_i - 1}$. It is guaranteed that all the $2n$ integers, $l_1, l_2, \dots, l_n, r_1, r_2, \dots, r_n$, are distinct.

You need to perform each operation exactly once, in arbitrary order.

You want to maximize the number of i $(0 \le i < 2n)$ such that $a_i \ne a_{i+1}$ after all n operations. Output the maximum number.

The original problem can be converted into a bipartite matching problem. Let $2n$ vertices denote the left and right endpoints of the n intervals. The right endpoint r_a of an interval a is connected to the left endpoint l_b of the other interval b if and only if r_a is in b and r_b is in a. It can be proved that the answer is equal to $2n$ minus the number of edges in the maximum matching. It can be solved in $O(n\sqrt{n})$ time by the Hopcroft-Karp algorithm [2] equipped with a segment tree for efficiently finding unvisited vertices.

In our benchmarks, we encode the matching problem and remove all but one right endpoints that are not matched in some fixed maximum matching. The benchmarks are UNSAT.

References

[1] ICPC Asia East Continent Final Contest 2022.
[2] J.E. Hopcroft and R.M. Karp, An n 2.5 algorithm for maximum matchings in bipartite graphs, SIAM J. Comput., 2, 1973, pp. 225–231.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Python Function Register Allocation Benchmarks

Andrew Haberlandt* and Harrison Green*
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
{ahaberla, harrisog}@cmu.edu
*Authors contributed equally

***Abstract*—We describe SAT benchmarks encoding the problem of register allocation for Python functions.**

I. Introduction

Our benchmarks represent graph coloring problems generated by simulating register allocation on individual Python functions. Although in practice register allocation is solved using greedy polynomial-time algorithms, they still serve as interesting, naturally-occurring graph coloring problems. We find that these coloring problems are quite challenging for modern solvers.

II. Benchmark Generation

Each of our benchmarks is a simulated register allocation problem. For each of 20 Python functions, we generate a graph coloring problem for the function's variable interference graph. More precisely, nodes represent variables in a Python function and edges connect variables whose live ranges overlap.

We only submit UNSATISFIABLE benchmarks, since all of the SATISFIABLE register allocation problems we generated were solvable in less than 1 second. For our submitted instances, the number of colors k was chosen such that each instance is not trivially solvable but solvable within 5000 seconds.

We used the live range analysis from Tensorflow's 'autograph' Python static analysis module [1], which allowed us to generate ASTs for arbitrary Python functions and analyze live ranges for each variable.

To identify realistic Python functions to use for our benchmarks, we consider all functions reachable from PySAT [2], a popular Python module for manipulating SAT formulas. The functions in our benchmark set are a combination of functions in PySAT and functions in Python standard library modules imported by PySAT. We include only functions with at least 15 variables.

III. Individual Benchmark Description

Table I displays the name of each Python function, along with the number of nodes and edges in the interference graph. We use the `color` tool from [3] to encode a coloring problem with k colors for each graph.

AH is supported by NSF Graduate Research Fellowship Grant No. DGE-2140739.

Name	# Nodes	# Edges	k
BZ2File_write	16	103	11
CNF_to_alien	21	169	11
CNF_to_alien	21	169	12
CNFPlus_from_fp	29	399	12
collections_namedtuple	63	1524	15
DecompressReader_read	19	167	12
FileObject_open	19	158	12
FileObject_open	19	158	13
GzipFile_close	17	121	11
LZMAFile___init__	33	455	14
LZMAFile_write	16	103	12
os_fwalk	24	268	12
posixpath__joinrealpath	27	328	13
posixpath_expanduser	28	292	14
StreamReader_readline	23	243	13
WCNF_from_fp	44	935	13
WCNF_from_fp	44	935	14
WCNF_to_alien	32	395	14
WCNFPlus_from_fp	49	1157	13
WCNFPlus_to_alien	37	541	14

TABLE I: List of benchmark instances derived from Python functions.

References

[1] "liveness.py in tensorflow's python static analysis module," https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/autograph/pyct/static_analysis/liveness.py, accessed: 2023-04-30.
[2] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: A Python toolkit for prototyping with SAT oracles," in *SAT*, 2018, pp. 428–437. [Online]. Available: https://doi.org/10.1007/978-3-319-94144-8_26
[3] M. J. Heule, A. Karahalios, and W.-J. van Hoeve, "From cliques to colorings and back again," in *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

A SAT-Benchmark Set from the Approximation of Trigonometric Functions for SAT-based Verification

Kai Hiller
Karlsruhe Institute of Technology
Karlsruhe, Germany
kai.hiller@student.kit.edu

Alexander Weigl
Karlsruhe Institute of Technology
Karlsruhe, Germany
weigl@kit.edu

I. Introduction

In this note, we present a benchmark set for SAT solvers, based on different approximations of trigonometric functions. This benchmark was inspired by the verification of trajectory planners used in automotive and robotic driving. Therefore, the benchmark origins from C programs, which are based on the floating point arithmetic. We use CBMC[1] for the translation into the DIMACS file format. This is a submission for the SAT Competition 2023. All benchmark files and sources can be found here:

https://gitlab.com/V02460/sat-bench-trig

In the following of this note, we show explain how to generate the DIMACS files using CBMC (Sect. II) and give performance insights (Sect. III).

II. Approximation of Trigonometric Function

We have implemented three different approximation of the sin and cos-function. Note, that you retrieve the other trigonometric functions by these equations:

$$\sin(x) = \cos(x - 90)$$
$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

All three functions are composed of a single partial cosine implementation in the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$. All functions are written in a single C file (given in the Appendix). You can activate an approximation by using defines (`-D`) on the command line. Also, some approximations allow to set the precision of the approximation. We have implemented:

- Bhaskara I[2] is a sine approximation formula, rewritten here to approximate the cosine:
$$\cos x \approx \frac{\pi^2 - 4x^2}{\pi^2 + x^2}, \qquad (-\frac{\pi}{2} \leq x \leq \frac{\pi}{2})$$
You can activate Bhaskara I with `-DTRIG_COS_BHASKARA`.
- With `-DTRIG_COS_LINES`, you activate a piece-wise linear approximation with pre-computed 5 or 17 line segments – activate with `-DTRIG_COS_LINE_SEGMENT_N=5` and `-DTRIG_COS_LINE_SEGMENT_N=17`.
- We also provide an approximation using the Taylor series expansion – activate with a `-DDTRIG_COS_TAYLOR` and the number of Taylor terms ($n = \{2, 4, 6\}$) with `-DTRIG_COS_TAYLOR_TERMS_N=n`.

The benchmark verifies a set of invariants that must hold for trigonometric functions. It asserts the functions for value range ($\cos\alpha, \sin\alpha \in [-1, 1]$), signedness on different intervals, some well-known values ($\pm\pi, \pm\frac{\pi}{2}, 0$), and the Pythagorean identity ($\cos^2\alpha + \sin^2\alpha = 1$).

To generate the CNF files with various variants of complexity, the following command is used:

```
cbmc <DEFINES> --dimacs --outfile <out>.cnf
```

In particular the submitted benchmark files are generated by the following commands:

```
bhaskara.cnf: -DTRIG_COS_BHASKARA
lines5.cnf:   -DTRIG_COS_LINES -DTRIG_COS_LINE_SEGMENT_N=5
lines17.cnf:  -DTRIG_COS_LINES -DTRIG_COS_LINE_SEGMENT_N=17
taylor2.cnf:  -DTRIG_COS_TAYLOR -DTRIG_COS_TAYLOR_TERMS_N=2
taylor4.cnf:  -DTRIG_COS_TAYLOR -DTRIG_COS_TAYLOR_TERMS_N=4
taylor6.cnf:  -DTRIG_COS_TAYLOR -DTRIG_COS_TAYLOR_TERMS_N=6
```

III. Performance

For an evaluation of this submission, we measured the provided DIMACS instances with KISSAT[3] 3.0.0 on AMD Ryzen 7 3700X CPU with 32 GB RAM. The instances were generated with CBMC 5.50.0. The run time and maximum set size as reported by kissat are provided in Table I.

TABLE I
Overview of the benchmark

Benchmark	Time [min]	Max. Set Size [MB]
`bhaskara`	03:01	334
`lines5`	00:10	114
`lines17`	00:13	157
`taylor2`	03:53	353
`taylor4`	19:18	592
`taylor6`	32:06	876

[1]https://www.cprover.org/cbmc/
[2]https://en.wikipedia.org/wiki/Bhaskara_I' s_sine_approximation_formula
[3]https://github.com/arminbiere/kissat

Appendix

A. Source Code

```
#include <assert.h>
#include <math.h>
#include <stdbool.h>
#include <stddef.h>


#define ARR_LEN(a) (sizeof(a)/sizeof(a[0]))
#define ARR_LAST(a) ((a)[ARR_LEN((a))-1])
#define ARR_FOREACH(a, body) for(size_t i=0; i<ARR_LEN((a)); i++) body
#define ARR_FOREACH_PAIR(a, body) for(size_t i=0, j=1; j<ARR_LEN((a)); i++, j++) body

bool is_between_open(float min, float x, float max) {
    return min < x && x < max;
}
bool is_between_closed(float min, float x, float max) {
    return min <= x && x <= max;
}
bool is_near(float x, float v, float e) {
    float d = x - v;
    return -e <= d && d <= e;
}

float nondet_float();
float nondet_between_open(float min, float max) {
    float r = nondet_float();
    __CPROVER_assume(is_between_open(min, r, max));
    return r;
}
float nondet_between_closed(float min, float max) {
    float r = nondet_float();
    __CPROVER_assume(is_between_closed(min, r, max));
    return r;
}


float cos_part(float x) {
    assert(is_between_closed(-M_PI_2, x, M_PI_2));

#ifdef TRIG_COS_TAYLOR
    float x2 = x * x;

    // Taylor series expansion.
    // Use an even number of taylor terms as their inaccuracies lead to f(PI/2) > 0.
    float r = 0.0;
    #if TRIG_COS_TAYLOR_TERMS_N >= 6
    r = x2 * (r + (1.0/479001600.0));
    r = x2 * (r - (1.0/3628800.0));
    #endif
    #if TRIG_COS_TAYLOR_TERMS_N >= 4
    r = x2 * (r + (1.0/40320.0));
    r = x2 * (r - (1.0/720.0));
    #endif
    #if TRIG_COS_TAYLOR_TERMS_N >= 2
    r = x2 * (r + (1.0/24.0));
    r = x2 * (r - (1.0/2.0));
    #endif

    return r + 1.0;

#elif defined(TRIG_COS_LINES)
    #if TRIG_COS_LINE_SEGMENT_N == 5
        #define _TRIG_COS_AS1 1.0054194
        #define _TRIG_COS_BS1 -0.12736896
        static const float as[] = {1.0, _TRIG_COS_AS1, 1.06930661, 1.19067734, 1.35429905, 1.52567402};
        static const float bs[] = {0.0, _TRIG_COS_BS1, -0.37707224, -0.61100686, -0.8158226, -0.96930506};
        static const float us[] = {(1.0 - _TRIG_COS_AS1) / _TRIG_COS_BS1, 0.25585252, 0.51882326,
            0.79887273, 1.11657691};
    #elif TRIG_COS_LINE_SEGMENT_N == 17
        #define _TRIG_COS_AS1 1.00071168
        #define _TRIG_COS_BS1 -0.04619989
        static const float as[] = {1.0, _TRIG_COS_AS1, 1.00921298, 1.02599767, 1.05063307, 1.08247781,
            1.12069107, 1.16424457, 1.2119373, 1.2624129, 1.31417934, 1.3656305, 1.4150707, 1.46073625,
            1.5008367, 1.53352255, 1.55715278, 1.56936604};
        static const float bs[] = {0.0, _TRIG_COS_BS1, -0.1382055, -0.22903199, -0.31790447, -0.40406469,
            -0.48677758, -0.56533745, -0.63907406, -0.70735831, -0.76960766, -0.82529087, -0.87393345,
            -0.91511821, -0.94850194, -0.97376936, -0.99081862, -0.99907977};
        static const float us[] = {(1.0 - _TRIG_COS_AS1) / _TRIG_COS_BS1, 0.09239975, 0.18479949,
            0.27719924, 0.36959898, 0.46199872, 0.55439845, 0.64679818, 0.7391979, 0.83159761, 0.9239973,
            1.01639697, 1.10879659, 1.20119615, 1.29359552, 1.38599475, 1.47839099};
    #else
        #error "No␣data␣for␣given␣number␣of␣line␣segments"
    #endif
    assert(ARR_LEN(as) == ARR_LEN(bs) && ARR_LEN(bs) == ARR_LEN(us) + 1);
    ARR_FOREACH_PAIR(us, {
        assert(us[i] < us[j]);
    })

    x = fabsf(x);

    float a = ARR_LAST(as);
    float b = ARR_LAST(bs);
    ARR_FOREACH(us, {
        if (x < us[i]) {
            a = as[i];
            b = bs[i];
            break;
        }
    })
    return a + b * x;

#else
    // https://en.wikipedia.org/wiki/Bhaskara_I%27s_sine_approximation_formula
    float pi2 = (float)M_PI * (float)M_PI;
    float x2 = x * x;
    return (pi2 - 4.0 * x2) / (pi2 + x2);

#endif
}

float flt_cos(float x) {
    assert(is_between_closed(-M_PI, x, M_PI));
    x = fabsf(x);
    return x <= M_PI_2 ? cos_part(x) : -cos_part(M_PI - x);
}

float flt_sin(float x) {
    assert(is_between_closed(-M_PI, x, M_PI));
    return x > 0.0 ? cos_part(x - M_PI_2) : -cos_part(x + M_PI_2);
}

float flt_tan(float x) {
    assert(is_between_open(-M_PI_2, x, M_PI_2));
    return flt_sin(x) / flt_cos(x);
}

int main() {
    float e = 0.1;

    float angle = nondet_between_closed(-M_PI, M_PI);
    float cos_r = flt_cos(angle);
    float sin_r = flt_sin(angle);

    // Check range
    assert(is_between_closed(-1.0, cos_r, 1.0));
    assert(is_between_closed(-1.0, sin_r, 1.0));

    // Check sign
    assert(flt_cos(nondet_between_open(-M_PI, -M_PI_2)) <= 0.0);
    assert(flt_cos(nondet_between_open(-M_PI_2, M_PI_2)) >= 0.0);
    assert(flt_cos(nondet_between_open(M_PI_2, M_PI)) <= 0.0);

    assert(flt_sin(nondet_between_open(-M_PI, 0.0)) <= 0.0);
    assert(flt_sin(nondet_between_open(0.0, M_PI)) >= 0.0);

    assert(flt_tan(nondet_between_open(-M_PI_2, 0.0)) <= 0.0);
    assert(flt_tan(nondet_between_open(0.0, M_PI_2)) >= 0.0);

    // Check values
    assert(is_near(flt_cos(-M_PI), -1.0, e));
    assert(is_near(flt_cos(-M_PI_2), 0.0, e));
    assert(is_near(flt_cos(0.0), 1.0, e));
    assert(is_near(flt_cos(M_PI_2), 0.0, e));
    assert(is_near(flt_cos(M_PI), -1.0, e));

    assert(is_near(flt_sin(-M_PI), 0.0, e));
    assert(is_near(flt_sin(-M_PI_2), -1.0, e));
    assert(is_near(flt_sin(0.0), 0.0, e));
    assert(is_near(flt_sin(M_PI_2), 1.0, e));
    assert(is_near(flt_sin(M_PI), 0.0, e));

    assert(is_near(flt_tan(-M_PI_2 * 0.5), -1.0, e));
    assert(is_near(flt_tan(0.0), 0.0, e));
    assert(is_near(flt_tan(M_PI_2 * 0.5), 1.0, e));

    // Pythagorean identity
    assert(is_near(cos_r * cos_r + sin_r * sin_r, 1.0, e));

    return 0;
}
```

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Benchmark Problems from Parameterized Encoding of Brent Equations over $\mathbb{Z}_2$

Karthikeya Namoju[1], Kalind Karia[2], Supratik Chakraborty[3], Biswabandan Panda[4]
[1,3,4]Department of Computer Science and Engineering, [2]Department of Electrical Engineering
Indian Institute of Technology, Bombay
karthikeyaiitb@gmail.com[1], kalind1610@gmail.com[2], {supratik[3], biswa[4]}@cse.iitb.ac.in

I. Introduction

Given a pair of $n \times n$ matrices A and B over an underlying ring of elements, consider the problem of multiplying them to form an $n \times n$ matrix C. For $1 \le i, j \le n$, the value of $C[i,j]$ is defined by $\sum_{k=1}^{n} A[i,k] \cdot B[k,j]$, where the summation and multiplication are interpreted over the underlying ring. This approach to multiplying A and B requires n^3 multiplications over the underlying ring to obtain all n^2 elements of C. An interesting question to ask is whether all elements of C can be computed using strictly less than n^3 multiplications. It turns out that in most cases, this is possible. For example, if $n = 2$, we know that Strassen's algorithm [2] allows computing C using 7 instead of $2^3 = 8$ multiplications in the underlying ring. Similarly, if $n = 3$, we can use Laderman's algorithm [1] to compute C using 23 instead of $3^3 = 27$ multiplications in the underlying ring. In general, it is interesting to ask whether k multiplications in the underlying ring are sufficient for computing all n^2 elements in the product of two $n \times n$ matrices.

The Brent equations [3] give a generic way of encoding the above problem. Specifically, let $M_l, 1 \le l \le k$ denote the product terms in the underlying ring that are meant be used in computing the n^2 elements of C. We assume each M_l is of the form $\left(\sum_{1\le i\le n}\sum_{1\le j\le n} \alpha^l_{i,j} * A[i,j]\right) \cdot \left(\sum_{1\le i\le n}\sum_{1\le j\le n} \beta^l_{i,j} * B[i,j]\right)$, where $\alpha^l_{i,j}$ and $\beta^l_{i,j}$ are (small) integers and $\alpha^l_{i,j} * A[i,j]$ (resp. $\beta^l_{i,j} * B[i,j]$) is short-hand for addition (in the underlying ring) of $\alpha^l_{i,j}$ copies of $A[i,j]$ (resp. $\beta^l_{i,j}$ copies of $B[i,j]$). Finally, each element $C[i,j]$ of the product matrix is obtained by computing a linear sum of the product terms, i.e. for $1 \le i,j \le n$, $C[i,j] = \sum_{l=1}^{k} \gamma^l_{i,j} * M_l$, where the $\gamma^l_{i,j}$ are (small) integers, and $\gamma^l_{i,j} * M_l$ is short-hand for addition (in the underlying ring) of $\gamma^l_{i,j}$ copies of M_l. A solution to the matrix multiplication problem using k multiplications in the underlying ring is then obtained by finding values of $\alpha^l_{i,j}, \beta^l_{i,j}$ and $\gamma^l_{i,j}$ such that the following equations are satisfied:

$$\sum_{l=1}^{k} \alpha^l_{x,y} . \beta^l_{u,v} . \gamma^l_{i,j} = \delta_{i,x} . \delta_{j,v} . \delta_{y,u}$$

for all $x, y, u, v, i, j \in \{1, \ldots n\}$. In the above equations, $\delta_{m,n}$ denotes the Kronecker delta function that evaluates to 1 if $m = n$, and evaluates to 0 otherwise.

It turns out that finding solutions to the Brent equations aren't that easy, even with help from computers. A first step towards solving them is to consider all operations over the ring $\mathbb{Z}_2$, where the multiplication operation is interpreted as logical AND, and the additional operation is interpreted as logical XOR. With this interpretation, Brent equations for any n, k (regardless of whether $k \ge n^3$) gives rise to a set of Boolean constraints. It is easy to see that for every solution to the Brent equations over the ring of real numbers, there is also a solution to the Brent equations over $\mathbb{Z}_2$, where both addition and subtraction (additive inverse) are interpreted as logical XOR, and multiplication is interpreted as logical AND. The converse is however not always true: a solution to the Brent equations over $\mathbb{Z}_2$ may not correspond to a solution to the same equations over the ring of real numbers. Interestingly, although we know that there is a trivial solution for the Brent equations over real numbers (and hence for the corresponding equations over $\mathbb{Z}_2$) when $k \ge n^3$, the set of Boolean constraints arising from encoding the Brent equations over $\mathbb{Z}_2$ are not that trivially shown to be satisfiable by modern SAT solvers even when $k \ge n^3$. This is most likely because state-of-the-art SAT solvers have no special heuristics to detect encoding of Brent equations.

Let $\varphi_{n,k}$ denote the CNF (conjunctive normal form) formula encoding the Brent equations over $\mathbb{Z}_2$, using the following auxiliary Boolean variables (also called Tseitin variables [4]). For convenience of exposition, we assume that k is an odd number ≥ 3 (this can easily be relaxed to allow even numbers as well).

- $t_{x,y,u,v,l}$ to denote $\alpha^l_{x,y} \wedge \beta^l_{u,v}$ for $1 \le x, y, u, v \le n$ and $1 \le l \le k$.
- $z_{x,y,u,v,i,j,l}$ to denote $\alpha^l_{x,y} \wedge \beta^l_{u,v} \wedge \gamma^l_{i,j}$ for $1 \le x, y, u, v, i, j \le n$ and $1 \le l \le k$.
- $s_{x,y,u,v,i,j,1,m}$ to denote $\bigoplus_{l=1}^{m} \left(\alpha^l_{x,y} \wedge \beta^l_{u,v} \wedge \gamma^l_{i,j}\right)$ for $1 \le x, y, u, v, i, j \le n$ and all $m \in \{2p+1 \mid 1 \le p \le \frac{k-1}{2}\}$

The relation between the auxiliary variables and $\alpha^l_{i,j}, \beta^l_{i,j}, \gamma^l_{i,j}$ variables are encoded using the following set of constraints:

$$\begin{array}{rcl}
t_{x,y,u,v,l} & \Leftrightarrow & \alpha^l_{x,y} \wedge \beta^l_{u,v} \\
z_{x,y,u,v,i,j,l} & \Leftrightarrow & t_{x,y,u,v,l} \wedge \gamma^l_{i,j} \\
s_{x,y,u,v,l,1,3} & \Leftrightarrow & z_{x,y,u,v,i,j,1} \oplus z_{x,y,u,v,i,j,2} \oplus z_{x,y,u,v,i,j,3} \\
s_{x,y,u,v,1,m} & \Leftrightarrow & s_{x,y,u,v,1,m-2} \oplus z_{x,y,u,v,i,j,m-1} \\
 & & \oplus z_{x,y,u,v,i,j,m}, \quad \forall m \in \{2p+1 \mid 2 \le p \le \frac{k-1}{2}\}
\end{array}$$

Each of the above constraints involves at most 4 Boolean variables, and is easily encoded as a CNF formula. The conjunction of these CNF formulas gives $\varphi_{n,k}$ as another CNF formula. Since there always exists a way to multiply matrices A and B using n^3 multiplications in the underlying ring, we know that $\varphi_{n,k}$ is satisfiable for every $k \geq n^3$. However, finding such a satisfying assignment is not always an easy task for a SAT solver. Furthermore, checking satisfiability of $\varphi_{n,k}$ where $k < n^3$, is highly non-trivial in general. For example, although we know that $\varphi_{3,23}$ is satisfiable, we do not know if $\varphi_{3,22}$ is satisfiable. Our attempts at trying to check satisfiability of $\varphi_{3,22}$ timed out after 10000 seconds even on using the topmost solvers from the SAT 2022 competition. The impreesive work of Heule et al [5] also couldn't determine if $\varphi_{3,22}$ is true, despite using several advanced streamlining heuristics.

Given the difficulty of checking satisfiability of $\varphi_{n,k}$ in general, we adopt a different strategy to arrive at benchmark instances that can be solved by our solver (and other powerful solvers) within 5000 seconds on the StarExec cluster. Specifically, we constrain the formula $\varphi_{n,k}$ obtained above in the following way. Let p be a floating point number in $[0, 1]$. For each i, j in $\{1, \ldots n\}$ and for each $l \in \{1, \ldots k\}$, we toss a coin with probability of heads as p to decide whether a constant $0/1$ value must be assigned to $\alpha^l_{i,j}$ (resp. $\beta^l_{i,j}$, $\gamma^l_{i,j}$), or if $\alpha^k_{i,j}$ (resp. $\beta^l_{i,j}$, $\gamma^l_{i,j}$) must be left unassigned. In case the coin toss decides that $\alpha^l_{i,j}$ (resp. $\beta^l_{i,j}$, $\gamma^l_{i,j}$) must be assigned a value, we use the strategy discussed below to assign a value to the variable.

Recall that Laderman [1] already gave a solution to the Brent equations for $n = 3$ and $k = 23$. This solution, when interpreted over the ring $\mathbb{Z}_2$, yields concrete $0/1$ values for $\alpha^l_{i,j}, \beta^l_{i,j}$ and $\gamma^l_{i,j}$ for $1 \leq i, j \leq 3$ and $1 \leq l \leq 23$. We take cue from these concrete values to assign values to $\alpha^l_{i,j}$ (resp. $\beta^l_{i,j}, \gamma^l_{i,j}$) in $\varphi_{n,k}$, if the coin toss alluded to above decides that the variable must be assigned a value. Specifically, we assign 1 to $\alpha^l_{i,j}$ (resp. $\beta^l_{i,j}, \gamma^l_{i,j}$) iff Laderman's solution also assigned 1 to the specific variable. Notice that with this assignment scheme, a variable can get assigned 0 in one of two ways: if either Laderman's solution assigned 0 to the variable or if the value of i or j exceeds 3 or that of l exceeds 23 (recall that Laderman's solution was for $n = 3$ and $k = 23$).

The above strategy of assigning values yields a partial assignment of $\alpha^l_{i,j}$, $\beta^l_{i,j}$ and $\gamma^l_{i,j}$ for $1 \leq i, j \leq n$ and $1 \leq l \leq k$. We use these partial assignments to simplify the following equivalences encoded in the formula $\varphi_{n,k}$.

- $t_{x,y,u,v,l} \Leftrightarrow \alpha^l_{x,y} \wedge \beta^l_{u,v}$ is simplified to $t_{x,y,u,v,l} \Leftrightarrow 0$ if either $\alpha^l_{x,y}$ or $\beta^l_{u,v}$ is assigned 0. If $\alpha^l_{x,y}$ is assigned 1 and $\beta^l_{u,v}$ is unassigned, then the equivalence is simplified to $t_{x,y,u,v,l} \Leftrightarrow \beta^l_{u,v}$, and analogously if $\alpha^l_{x,y}$ is unassigned and $\beta^l_{u,v}$ is assigned 1. Finally, if both $\alpha^l_{x,y}$ and $\beta^l_{u,v}$ are assigned 1, we use the equivalence $t_{x,y,u,v,l} \Leftrightarrow 1$.
- $z_{x,y,u,v,i,j,l} \Leftrightarrow t_{x,y,u,v,l} \wedge \gamma^l_{i,j}$ is simplified to $z_{x,y,u,v,i,j,l} \Leftrightarrow 0$ if $\gamma^l_{i,j}$ is assigned 0. On the other hand, if $\gamma^l_{i,j}$ is assigned 1, then the simplified equivalence is

  $z_{x,y,u,v,i,j,l} \Leftrightarrow t_{x,y,u,v,l}$

Note that we do not propagate the partial assignment beyond the above two identities when constructing the constrained version of the formula $\varphi_{n,k}$.

In view of the above discussion, the final CNF formula obtained for our benchmark suite can have three potential parameters: n, k and p. For purposes of our submission, however, we fixed n to 3, and varied k and p. Each benchmark thus generated is named `brent_k_p.cnf`, where k is a positive integer and p is a floating point number in $[0, 1]$. These benchmarks and the satisfiability status as determined by our solver are given in the table below.

brent_65_0.cnf	sat
brent_51_0.29.cnf	sat
brent_51_0.07.cnf	sat
brent_9_0.cnf	unsat
brent_69_0.05.cnf	sat
brent_65_0.1.cnf	sat
brent_51_0.28.cnf	sat
brent_13_0.1.cnf	unsat
brent_63_0.1.cnf	sat
brent_63_0.2.cnf	sat
brent_67_0.05.cnf	sat
brent_69_0.cnf	sat
brent_71_0.25.cnf	sat
brent_63_0.22.cnf	sat
brent_63_0.26.cnf	sat
brent_51_0.17.cnf	sat
brent_15_0.25.cnf	unsat
brent_63_0.cnf	sat
brent_63_0.15.cnf	sat
brent_69_0.3.cnf	sat

TABLE I
20 BENCHMARKS LIST

Acknowledgements: The key ideas in the encoding discussed above are adapted from [5]. We thank Vijay Ganesh and Curtis Bright for discussions and comments on encoding the Brent equations over $\mathbb{Z}_2$ as Boolean formulas.

REFERENCES

[1] J. D. Laderman. A noncommutative algorithm for multiplying 3×3 matrices using 23 multiplications. Bulletin of the American Mathematical Society, 82(1):126–128, 1976.

[2] V. Strassen. Gaussian elimination is not optimal. Numerische Mathematik, 13(4):354–356, 1969.

[3] R. P. Brent. Algorithms for matrix multiplication. Technical report, Department of Computer Science, Stanford, 1970.

[4] G. S. Tseitin. On the complexity of derivation in propositional calculus. Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics, pages 115–125, 1968.

[5] M. Heule, M. Kauers and M. Seidl. Local Search for Fast Matrix Multiplication. Proceedings of SAT Conference, pg 155-163, 2019.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Testing the ASCON Hash Function

Norbert Manthey
nmanthey@conp-solutions.com
Dresden, Germany

I. Introduction

Hash functions have several properties. In times of quantum computing, cryptographic functions and their implementations have to meet specific security properties. The ASCON encryption algorithm family has been selected as meeting all relevant properties. Several analysis have been performed on these algorithms already [2].

We used searching for missing input bytes of a known partial plain text. First, we generate a random input, and compute the hash value related to this input. Next, for satisfiable formulas, we simply ask the SAT solver to find the bits for a random subset of the input bytes. For unsatisfiable formulas, we furthermore exchange one of the output bytes.

To simplify generating the relevant formulas, we use the ASCON reference implementation of the hash function, optimized for 64 bit CPUs. The library is included into the generating C program, leaving out the initializations that have to be found by the SAT solver. The expected hash value is then added as an assertion. Finally, the CBMC [1] tool converts such a C program into CNF. For each generated CNF, we generated the relevant C program before; as well as computed the actual hash value for the random input. For this conversion, CBMC in version 5.10 (cbmc-5.10) has been used.

II. Formula Selection

CNFs have been generated for formulas with four to thirteen input bytes. From those, one to three input bytes have been allowed to be guessed by the SAT solver – for each combination we generated one random combination of dropped bytes. Finally, for each configuration furthermore a likely unsatisfiable formula has been generated.

As MINISAT 2.2 is not the most recent SAT solver, and to not move further into a KISSAT solver mono culture, we filtered the generated formulas and dropped the ones that could be solved easily with KISSAT (version rel-3.0.0-1-g97917dd).

III. Availability

The source of the tool is publicly available at https://github.com/conp-solutions/ascon-c/tree/satcomp-2023. The repository also contains a script to generate a first set of formulas.

References

[1] E. Clarke, D. Kroening, and F. Lerda, “A tool for checking ANSI-C programs,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[2] D. Gérault, T. Peyrin, and Q. Q. Tan, “Exploring differential-based distinguishers and forgeries for ASCON,” *IACR Trans. Symmetric Cryptol.*, vol. 2021, no. 3, pp. 102–136, 2021. [Online]. Available: https://doi.org/10.46586/tosc.v2021.i3.102-136

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Symmetry Reduced SAT Encodings for the Social Golfer Problem

Shubh Jaju
Indian Institute of Technology Delhi
New Delhi, India
jajushubh.iitd@gmail.com

Valentin Mayer-Eichberger
Universität Potsdam
Potsdam, Germany
valentin@mayer-eichberger.de

Abdallah Saffidine
The University of New South Wales
Sydney, Australia
Abdallah.Saffidine@gmail.com

***Abstract*—We re-visit the *Social Golfer Problem* (SGP) that has received considerable attention in the SAT, constraint programming (CP) and local search communities because of its intriguing combinatorial structure with many symmetries. In our benchmark for the SAT competition we introduce a new SAT model for this problem using a different representation of a schedule. Our model reduces symmetries by using a context and table-free representation.**

I. Introduction

In 1850, Reverend Thomas Kirkman sent a query to the readers of a popular math magazine, Lady's and Gentleman's Diary:

> Fifteen young ladies in a school walk out 3 abreast for seven days in succession: it is required to arrange them daily, so that no two will walk twice abreast.

Finding such schedules still puzzles researchers today. In the constraint programming and SAT community, the generalised problem is known as the Social Golfer Problem with parameters $g - p - w$: How to schedule $g \cdot p$ players in g groups of size p for w weeks such that no players meet more than once.

The following table is a solution for the instance $4 - 3 - 4$: 12 players in 4 groups of 3 scheduled for four weeks. Each matrix is the schedule of one week, and each row in a matrix denotes one group.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} \begin{pmatrix} 1 & 5 & 7 \\ 2 & 9 & 10 \\ 3 & 4 & 12 \\ 6 & 8 & 11 \end{pmatrix} \begin{pmatrix} 1 & 4 & 10 \\ \mathbf{2} & \mathbf{5} & 8 \\ 3 & 7 & 11 \\ 6 & \mathbf{9} & \mathbf{12} \end{pmatrix} \begin{pmatrix} 1 & 8 & 12 \\ 2 & 6 & 7 \\ 3 & 5 & 10 \\ 4 & 9 & 11 \end{pmatrix}$$

II. The Approach

The typically SAT and CP representation of SGP is based on the variables $\mathrm{table}_{i,j,k,p}$ meaning in week i in group j plays player k at position p in this group. Research has focused on improving this representation and the required constraints, and applying symmetry breaking [1] [6] [3].

Our encoding works on a different set of variables which avoids an explicit representation of the schedule:

- $\mathrm{context}_{i,j,k,l}$ is true if player i and j play together in the same week in that k and l player together.

We describe the complete schedule just by the information which pair of players play together in the same week that another pair of players play together. This is a unique representation of the schedule and removes group and week symmetries by definition. The challenge here is to avoid a blow-up in the number of variables and clauses.

In the above example of the solution to $4-3-4$, the variable $\mathrm{context}_{2,5,9,12}$ is true because players 2 and 5 play in the same week (here week 3) together as players 9 and 12 play together. Note that we do not need an explicit representation of week 3.

The following set of constraints needs to be encoded in SAT:

- Fixing i and j, each player needs to occur exactly once as k or l.
- Players play at most once together.
- Transitive closure of group relations: If i and j play together, and j and k play together in one group, then also i and k.

A naive encoding of these constraints leads to too many clauses and variables. The following set of techniques were used to overcome this challenge.

- We require $i < j$ and $k < l$ for the variables $\mathrm{context}_{i,j,k,l}$.
- We fix one player in $\mathrm{context}_{i,j,k,l}$. In our encoding, we only consider j equal to the last player.
- We use the counter encoding to encode the cardinality constraint of group size.

The idea behind this approach was mentioned in the work of Barbara Smith in [4]. However, the idea was given with the comment that it would be very hard to formulate these constraints in practice, and it was not implemented in her work. To the best of our knowledge, we are the first to attempt a concrete implementation.

III. The Benchmark

There is a natural upper bound of the number of weeks given the number of players and groups since, at some point, a player will have to meet another player again to make a schedule. Finding schedules is easy for a few weeks and becomes very hard close to this upper bound. Unfortunately, just a few instances are not too easy but not too hard for SAT solvers. The benchmark consists of the instances we could find to be interesting. Furthermore, to complete the set of instances we have also added instances from the standard table encoding as a comparison. All instances in this benchmark are satisfiable. The benchmark has been prepared with the SAT programming language Bule [2] which provides a DIMACS output.

We propose the satisfiable instance of $8 - 4 - 10$ as a milestone for SAT. Other incomplete and greedy methods have solved this problem [5], but we are unaware of either SAT or CP to successfully compute a schedule. We believe a combination of encoding and solving techniques is necessary to achieve this goal.

REFERENCES

[1] Gent, I.P., Lynce, I.: A sat encoding for the social golfer problem. Modelling and Solving Problems with Constraints **2** (2005)

[2] Jung, J.C., Mayer-Eichberger, V., Saffidine, A.: QBF programming with the modeling language bule. In: Meel, K.S., Strichman, O. (eds.) SAT 2022. LIPIcs, vol. 236 (2022)

[3] Lardeux, F., Monfroy, E., Rodriguez-Tello, E., Crawford, B., Soto, R.: Solving complex problems using model transformations: from set constraint modeling to sat instance solving. Expert Systems with Applications p. 113243 (01 2020). https://doi.org/10.1016/j.eswa.2020.113243

[4] Smith, B.M.: Reducing symmetry in a combinatorial design problem. In: Proceedings of the Third International Workshop on Integration of AI and OR Techniques. p. 351–359 (2001)

[5] Triska, M., Musliu, N.: An effective greedy heuristic for the social golfer problem. Annals of Operations Research **194**(1), 413–425 (2012)

[6] Triska, M., Musliu, N.: An improved sat formulation for the social golfer problem. Annals of Operations Research **194**(1), 427–438 (2012)

SAT Encodings of Acceptance Problems in Abstract Argumentation

Andreas Niskanen
Helsinki Institute for Information Technology HIIT,
Department of Computer Science,
University of Helsinki, Finland
Email: andreas.niskanen@helsinki.fi

***Abstract*—Abstract argumentation is one of the major approaches to formal argumentation with various applications in artificial intelligence. In an abstract argumentation framework (AF), arguments are modeled as vertices in a directed graph, with edges representing attacks between arguments. Different semantics map such an AF to a set of extensions, that is, jointly acceptable sets of arguments. An individual argument is then accepted if it is contained in one or all extensions. We briefly describe benchmarks for SAT encodings of such acceptance problems.**

***Index Terms*—abstract argumentation, credulous acceptance, skeptical acceptance**

I. Abstract Argumentation

We recall argumentation frameworks and their semantics [1]. An *argumentation framework (AF)* is a pair $F = (A, R)$, where A is a (finite) set of arguments, and $R \subseteq A \times A$ is an attack relation. If $(a, b) \in R$, we say that argument a attacks argument b. An argument $a \in A$ is *defended* by a set $S \subseteq A$ if, for each $b \in A$ with $(b, a) \in R$, there is a $c \in S$ with $(c, b) \in R$. A set $S \subseteq A$ is *conflict-free* if there is no $a, b \in S$ with $(a, b) \in R$. We denote the collection of conflict-free sets of F by $cf(F)$.

Semantics map each AF $F = (A, R)$ to a set $\sigma(F) \subseteq 2^A$ of extensions. We consider for σ the functions *adm* and *stb*, which stand for *admissible* and *stable* semantics, respectively. For a conflict-free set $S \in cf(F)$, it holds that $S \in adm(F)$ if each $a \in S$ is defended by S, and $S \in stb(F)$ if for each $a \in A \setminus S$, there is some $b \in S$ with $(b, a) \in R$. Finally, an argument $q \in A$ is *credulously accepted* under σ if there is an extension $E \in \sigma(F)$ with $q \in E$, and *skeptically accepted* under σ if for all extensions $E \in \sigma(F)$ it holds that $q \in E$.

II. SAT Encodings

We briefly outline SAT encodings of admissible and stable semantics [2]. Let $F = (A, R)$ be an AF. To represent an extension, for each argument $a \in A$, we introduce a variable x_a. Admissible sets are encoded as $\varphi_{adm}(F) = \bigwedge_{(a,b)\in R}(\neg x_a \vee \neg x_b) \wedge \bigwedge_{a\in A}(x_a \rightarrow \bigwedge_{(b,a)\in R} \bigvee_{(c,b)\in R} x_c)$, and stable semantics as $\varphi_{stb}(F) = \varphi_{adm}(F) \wedge \bigwedge_{a\in A}(x_a \vee \bigvee_{(b,a)\in R} x_b)$. Now, for $\sigma \in \{adm, stb\}$, an argument $q \in A$ is credulously accepted under σ iff $\varphi_\sigma(F) \wedge x_q$ is satisfiable, and skeptically accepted under σ iff $\varphi_\sigma(F) \wedge \neg x_q$ is unsatisfiable.

This work is supported by Academy of Finland (grant #347588).

III. Benchmarks

We used 23 different AFs from the ICCMA 2017 competition [3], benchmark set B (http://argumentationcompetition.org/2017/B.tar.gz). We selected all instances from the "too hard" category from the Erdös-Rényi, Watts-Strogatz, and StableGenerator domains. Finally, we considered both query arguments provided in the benchmark set (`.arg1` and `.arg2`), and three different acceptance tasks, namely credulous acceptance under admissible (DC-AD) and stable semantics (DC-ST), and skeptical acceptance under stable semantics (DS-ST). This resulted in $23 \cdot 2 \cdot 3 = 138$ instances. The CNF encodings were generated using the SAT-based AF solver μ-TOKSIA [4].

The instances are named according to the format `<af>_<query>_<task>.cnf` where `<af>` is the original AF instance, `<query>` is the query argument used (1 or 2, which stand for files `<af>.arg1` or `<af>.arg2` in the benchmark data set), and `<task>` is the acceptance task (`DC-AD`, `DC-ST`, `DS-ST`). Note that all AF instances and thus all file names start with either `ER`, `WS`, or `stb`. These identifiers correspond to the graph generation model (Erdös-Rényi, Watts-Strogatz, and StableGenerator, respectively).

References

[1] P. M. Dung, "On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games," *Artif. Intell.*, vol. 77, no. 2, pp. 321–358, 1995.
[2] P. Besnard and S. Doutre, "Checking the acceptability of a set of arguments," in *10th International Workshop on Non-Monotonic Reasoning (NMR 2004), Whistler, Canada, June 6-8, 2004, Proceedings*, J. P. Delgrande and T. Schaub, Eds., 2004, pp. 59–64.
[3] S. A. Gaggl, T. Linsbichler, M. Maratea, and S. Woltran, "Design and results of the second international competition on computational models of argumentation," *Artif. Intell.*, vol. 279, 2020.
[4] A. Niskanen and M. Järvisalo, "μ-toksia: An efficient abstract argumentation reasoner," in *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020*, D. Calvanese, E. Erdem, and M. Thielscher, Eds., 2020, pp. 800–804.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Crafted Benchmark Formulas Requiring Symmetry Breaking and/or Parity Reasoning

Bart Bogaerts
Vrije Universiteit Brussel
bart.bogaerts@vub.be

Jakob Nordström
University of Copenhagen and Lund University
jn@di.ki.dk

Andy Oertel
Lund University and University of Copenhagen
andy.oertel@cs.lth.se

Çağrı Uluç Yıldırımoğlu
Vrije Universiteit Brussel
cagri.uluc.yildirimoglu@vub.be

***Abstract*—We propose a benchmark set containing crafted CNF formulas for which the hardness with respect to resolution scales exponentially with the formula size, which means that these formulas will be very challenging for CDCL SAT solvers without strong pre- or inprocessing. However, the formulas are designed so that different formulas in the benchmark set are amenable to symmetry breaking, parity reasoning, or a combination of these two techniques, respectively. In this way, our benchmarks can be used to evaluate to what extent solvers are able to implement such techniques efficiently.**

I. Introduction

We contribute a set of unsatisfiable crafted CNF formulas that are designed to be exponentially hard for the resolution proof system [6], [12], [11], [17], and hence also for conflict-driven clause learning (CDCL) SAT solvers, which decide unsatisfiability of CNF formulas by, in effect, constructing resolution proofs of unsatisfiability [3]. However, if resolution-based reasoning is combined with symmetry breaking, parity reasoning, or sometimes a combination of these two techniques, then the formulas become trivial from a theoretical point of view. Our goal is that these benchmark formulas could be used to evaluate to what extent SAT solvers can efficiently perform symmetry breaking or parity reasoning in practice.

We provide benchmark instances from five different formula families, namely clique-colouring formulas (Section II), Tseitin formulas (Section III), relativized pigeonhole principle (RPHP) formulas (Section IV), graph ordering principle formulas with XOR substitution (Section V), and random 3-XOR formulas with OR substitution (Section VI). All contributed formulas are unsatisfiable, since this is the setting in which the theoretical hardness guarantees apply.

All formulas were generated using CNFGEN [15], [9], except for the relativized pigeonhole principle instances where a dedicated C program was employed.

II. Clique-Colouring Formulas

The *clique-colouring formula* with parameters n, k, and c encodes the claim that there exists a graph on n vertices which has a clique of size k and is simultaneously c-colourable. This is a contradiction if $k > c$. In our benchmark instances we fix $k = c + 1$.

Pudlák [16] proved that if $k = c + 1$ scales like $\sqrt[4]{n}$, then clique-colouring formulas are exponentially hard not only for resolution but even for the exponentially stronger cutting planes proof system [10]. However, if symmetry reasoning is used, then we can argue that without loss of generality there is a k-clique on the first k vertices, after which a second symmetry argument says that these k vertices have to be coloured with colours $1, 2, \ldots, c$ until we run out of colours.

The clique-colouring formula instances in our benchmark set have file names with the prefix `cliquecolouring`. Since Pudlák's result only provides asymptotic lower bounds, we have tried to tune $k = c + 1$ with respect to n to get suitably small formulas that exhibit interesting properties in practice.

III. Tseitin Formulas

A *Tseitin formula* over a graph G is, very roughly speaking, a convoluted way of encoding the handshaking lemma, namely that the sum of all vertex degrees in an undirected graph must be an even number. These formulas have received their name from Tseitin [19], and Urquhart [20] proved that Tseitin formulas are exponentially hard for resolution if G is a very well-connected bounded-degree expander graph.[1]

In a bit more detail, for a graph G and a charge function $\chi : V(G) \to \{0, 1\}$ such that the sum $\sum_{v \in V(G)} \chi(v)$ over all vertices is odd, the Tseitin formula for G and χ has a variable x_e for every edge $e \in E(G)$. For every vertex $v \in V(G)$ the formula contains the parity constraint $\sum_{e \in E(v)} x_e = \chi(v) \pmod 2$ encoded in CNF, where $E(v)$ is the set of edges incident to v in G. The size of Tseitin formulas scale linearly with the number of vertices if the vertex degrees are bounded by some (universal) constant.

Since Tseitin formulas are a special case of inconsistent systems of linear equations $\bmod 2$, they can easily be refuted by solvers doing parity reasoning using Gaussian elimination. Less obviously, Tseitin formulas are also easy to solve by symmetry breaking, since every cycle in G induces a symmetry where all variables/edges in the cycle can have their values flipped simultaneously. This can be used to eliminate cycles one by one in the graph by fixing some edge in every cycle to 0 and removing it. This eventually reduces the graph to a tree, for which the Tseitin formula is solved by unit propagation.

We have included two subfamilies of Tseitin formulas in our benchmark set that have varying difficulty for solvers that

[1]Quite confusingly, and probably because of Urquhart's paper [20], Tseitin formulas over expander graphs are sometimes referred to in the SAT solving community as "Urquhart formulas," which does not appear to be established terminology elsewhere.

do not implement parity reasoning or symmetry breaking.

A. Tseitin Formulas over Grid Graphs

The vertices in a grid graph form an $n \times m$ grid, where each vertex is adjacent to its four neighbours horizontally and vertically on the grid. The size of resolution refutations of such formulas scale exponentially with the smaller grid dimension, which is at most exponential in the square root of the formula size.

Our Tseitin formulas over grid graphs have filenames with the prefix `tseitin_grid`.

B. Tseitin Formulas over Random 3-Regular Graphs

We also generate Tseitin formulas over random 3-regular graphs, for which resolution refutations asymptotically almost surely have size growing exponentially measured in the formula size.[2]

The Tseitin formula benchmark files generated from random 3-regular graphs have names starting with `tseitin_d3`.

IV. Relativized Pigeonhole Principle Formulas

The *relativized pigeonhole principle (RPHP) formula* with parameters p and r encodes, roughly speaking, that there are mappings $m_1 : [p] \to [r]$ and $m_2 : [r] \to [p-1]$ such that their composition $m_1 \circ m_2$ is injective. Such formulas have been studied in [2], [1].

In a bit more detail, and expressed in terms of pigeons and pigeonholes, the RPHP formula states that

1) First, the p pigeons should fly to r resting places, where each resting place can only be occupied by a single pigeon.
2) Every resting place contains the address of a final pigeonhole, and after resting the pigeon should continue to that hole.
3) The map from resting places to pigeonholes need not be injective in general, but it should be injective when restricted to the resting places actually used by the pigeons, so that at the final destination every pigeon gets its own pigeonhole.

Since there are p pigeons but only $p-1$ holes, this formula is just as unsatisfiable as the standard pigeonhole principle formula. What is interesting, however, is that if one fixes p to be a constant and lets r grow, then the minimal size of resolution refutations for such formulas scales roughly like r^p [1], which can be chosen to be a suitably large polynomial by fixing an appropriate constant p. That is, such formulas are "easy" in the theoretical sense of having polynomial-size refutations, but such refutations will grow too quickly for SAT solvers to be able to find them efficiently in practice.

By repeated use of symmetry reasoning, one can argue that all pigeons, resting places, and pigeonholes are interchangeable. This makes the formulas easy for a solver with strong enough symmetry breaking.

[2]There are also explicit constructions of good enough graphs that are guaranteed to yield exponentially hard formulas, but in practice random graphs will always satisfy the required properties and are much simpler to generate.

The relativized pigeonhole principle formulas in our benchmark set have filenames with prefixes `rphp`.

V. XORified Graph Ordering Principle Formulas

The *ordering principle formula* with parameter n encodes the contradictory claim that there exists a set with n elements that is partially ordered and yet does not contain any minimal element. This final condition is encoded for every element j by a clause saying that some other element $i \in \{1, 2, \ldots, n\} \setminus j$ is smaller.

Ordering principle formulas were studied by Krishnamurthy [14], who conjectured them to be exponentially hard for resolution. Stålmarck [18] showed that the formulas in fact have resolution refutations that are linear in the formula size, but the formulas are still quite interesting in that for CDCL solvers with the standard VSIDS decision heuristic performance is quite sensitive to the VSIDS decay factor [13].

The *graph ordering principle formula* is a more constrained version, in which a d-regular graph G (for $d \geq 3$) is defined over the n elements, and the fact that the element j is not minimal has to be witnessed by some neighbour $i \in N_G(j)$. This turns the encoding into a d-CNF formula, but if G is a so-called expander graph (which will hold asymptotically almost surely for a randomly generated d-regular graph),[3] then resolution refutations of graph ordering principle formulas need clauses of width linear in n [7] (i.e., clauses containing a number of literals that is linear in n).

From any CNF formula F we can generate a new formula $F[\oplus]$ by applying *XOR substitution*, or *XORification*, which is to replace every variable x by an exclusive or of two new, fresh variables $x_1 \oplus x_2$ and then expand the resulting formula to CNF in the canonical way. For k-CNF formulas of constant width k, this does not increase formula size more than by a constant factor 2^k. Ben-Sasson [4] (crediting Alekhnovich and Razborov) observed that if a CNF formula F requires resolution refutations of width w, then $F[\oplus]$ requires resolution refutations of size exponential in w. Our *XORified graph ordering principle formulas*, which are generated from graph ordering principle formulas by XOR substitution, are therefore exponentially hard for resolution (at least asymptotically almost surely).

By using strong enough symmetry reasoning (or potentially by introducing a new variable x that is defined to be the value of $x_1 \oplus x_2$ for every pair of variables resulting from XORification), it is possible to "un-XORify" $F[\oplus]$ and recover the original formula F, after which the short resolution refutation of F can be applied. This means that solvers that are able to perform such reasoning should also be able to decide unsatisfiability of XORification graph ordering principle formulas efficiently.

For our XORified graph ordering principle formulas in our benchmark set we used 3-regular graphs, and the formulas have file names with the prefix `xor_op`.

[3]Again, there are also explicit constructions of such graphs, but in practice random graphs always work.

VI. ORIFIED RANDOM 3-XOR FORMULAS

A random k-XOR formula with m constraints over n variables is generated by randomly selecting m times a set of k variables $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ for $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ and a bit $b \in \{0, 1\}$ and then adding the parity constraint $x_{i_1} \oplus x_{i_2} \oplus \cdots \oplus x_{i_k} = b$ encoded in CNF to the formula. If k is constant, then the size of this formula scales linearly with m.

For random 3-XOR formulas with $m = n$ it holds asymptotically almost surely as n goes to infinity that the formulas are unsatisfiable and require resolution refutations of exponential size (which can be established by techniques as in [5]). However, any unsatisfiable XOR formula (random or not) is trivial to refute by doing Gaussian elimination over GF(2), which means that random 3-XOR formulas are easy for solvers that implement parity reasoning in this way.

From any CNF formula F we can generate a new formula $F[\vee]$ by applying *OR substitution*, or *ORification*, where every variable x is replaced by a standard or $x_1 \vee x_2$ of two new, fresh variables, after which the result is expanded to CNF again in the canonical way. For ORified versions of 3-XOR formulas Gaussian elimination no longer works, since the parity constraints have been destroyed, but the formulas still have small refutations in the more general polynomial calculus proof system [8] formalizing Gröbner basis computations.

Analogously to what holds for the XORified formulas in Section V, one can use (human) symmetry reasoning or introductions of new auxiliary variables to "un-ORify" ORified 3-XOR formulas, after which parity reasoning can be used to derive contradiction. This means that at least in theory, a solver that combined strong enough symmetry reasoning or variable introduction with parity reasoning—or, alternatively, implemented a strong enough version of general algebraic reasoning, as noted in the previous paragraph—could solve these formulas easily.

Our ORified random 3-XOR formulas have filenames starting with `or_randxor`.

REFERENCES

[1] A. Atserias, M. Lauria, and J. Nordström, "Narrow proofs may be maximally long," *ACM Transactions on Computational Logic*, vol. 17, no. 3, pp. 19:1–19:30, May 2016, preliminary version in *CCC '14*.

[2] A. Atserias, M. Müller, and S. Oliva, "Lower bounds for DNF-refutations of a relativized weak pigeonhole principle," *Journal of Symbolic Logic*, vol. 80, no. 2, pp. 450–476, Jun. 2015, preliminary version in *CCC '13*.

[3] P. Beame, H. Kautz, and A. Sabharwal, "Towards understanding and harnessing the potential of clause learning," *Journal of Artificial Intelligence Research*, vol. 22, pp. 319–351, Dec. 2004, preliminary version in *IJCAI '03*.

[4] E. Ben-Sasson, "Size-space tradeoffs for resolution," *SIAM Journal on Computing*, vol. 38, no. 6, pp. 2511–2525, May 2009, preliminary version in *STOC '02*.

[5] E. Ben-Sasson and A. Wigderson, "Short proofs are narrow—resolution made simple," *Journal of the ACM*, vol. 48, no. 2, pp. 149–169, Mar. 2001, preliminary version in *STOC '99*.

[6] A. Blake, "Canonical expressions in Boolean algebra," Ph.D. dissertation, University of Chicago, 1937.

[7] M. L. Bonet and N. Galesi, "Optimality of size-width tradeoffs for resolution," *Computational Complexity*, vol. 10, no. 4, pp. 261–276, Dec. 2001, preliminary version in *FOCS '99*.

[8] M. Clegg, J. Edmonds, and R. Impagliazzo, "Using the Groebner basis algorithm to find proofs of unsatisfiability," in *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC '96)*, May 1996, pp. 174–183.

[9] "CNFgen: Combinatorial benchmarks for SAT solvers," https://massimolauria.net/cnfgen/.

[10] W. Cook, C. R. Coullard, and G. Turán, "On the complexity of cutting-plane proofs," *Discrete Applied Mathematics*, vol. 18, no. 1, pp. 25–38, Nov. 1987.

[11] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962.

[12] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[13] J. Elffers, J. Giráldez-Cru, S. Gocht, J. Nordström, and L. Simon, "Seeking practical CDCL insights from theoretical SAT benchmarks," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18)*, Jul. 2018, pp. 1300–1308.

[14] B. Krishnamurthy, "Short proofs for tricky formulas," *Acta Informatica*, vol. 22, no. 3, pp. 253–275, Aug. 1985.

[15] M. Lauria, J. Elffers, J. Nordström, and M. Vinyals, "CNFgen: A generator of crafted benchmarks," in *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT '17)*, ser. Lecture Notes in Computer Science, vol. 10491. Springer, Aug. 2017, pp. 464–473.

[16] P. Pudlák, "Lower bounds for resolution and cutting plane proofs and monotone computations," *Journal of Symbolic Logic*, vol. 62, no. 3, pp. 981–998, Sep. 1997.

[17] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM*, vol. 12, no. 1, pp. 23–41, Jan. 1965.

[18] G. Stålmarck, "Short resolution proofs for a sequence of tricky formulas," *Acta Informatica*, vol. 33, no. 3, pp. 277–280, May 1996.

[19] G. Tseitin, "On the complexity of derivation in propositional calculus," in *Structures in Constructive Mathematics and Mathematical Logic, Part II*, A. O. Silenko, Ed. Consultants Bureau, New York-London, 1968, pp. 115–125.

[20] A. Urquhart, "Hard examples for resolution," *Journal of the ACM*, vol. 34, no. 1, pp. 209–219, Jan. 1987.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Subsumption Benchmarks

Luke Nuttall
Independent
Yantai, China
lukerossnuttall@gmail.com

***Abstract*—Benchmarks were made to test solvers which either explicitly or heuristically / implicitly perform self-subsuming resolution of clauses.**

Keywords—benchmark, subsumption, resolution

I. Introduction

Many different solvers have been submitted for the SAT competition using a variety of methods, which are only usually equivalent. Some techniques will not explicitely perform techniques used by others, but will heuristically produce similar results. For exampe, any process that continually produces resolutions of clauses will eventually make a groebner basis of the solution space, even though that wasn't the aim.

The purpose of these benchmarks is to determine if the techniques employed by solvers will heuristically approximate a simple but often wasteful technique – subsumption.

II. subsumption

A. Clause subsumption

Given a problem instance with the two CNF clauses; {a,b,c & a,b} the first clause is redundant of the second clause. Any solution which satisfies the second clause implies that the first is also satisfied (it doesn't matter what c is). The second clause *subsumes* the first clause, which can be removed from the instance without changing the solution set.

B. Self-subsuming resolution

Given a problem instance with the two CNF clauses; {a,b,c & a,-c} neither clause subsumes the other, but the resolution of the two clauses a,b subsumes the first.

Self-subsuming resolution is the technique of replacing the clause a,b,c with the clause a,b when it would not change the solution set (ie in light of the clause a,-c). This simplifies an applicable problem instance.

III. Worth trying?

Alone, this technique isn't going to solve any problems other than those specially crafted for this benchmark. Determining if a clause can be subsumed, either directly or through a resolution takes time which (under a developer's opinion) might be better spent performing other tasks.

The naive implementation requires O(n^2) tests of every clause agaisnt every other. Reducing the cost requires developing structures based on the variables in each clause; which again might not be 'worth it' for some solvers which need to generate incompatible structures.

This raises the question, what other techniqes can be used to ensure all subsumptions are replaced, to what degree of efficiency, and what techniques are fundamentally incompatible?

IV. The benchmarks

A set of benchmarks is made to test for solvers which effectively perform subsumption. A solver which only performs this one technique and no other would solve the benchmarks given. However, the benchmarks are large enough that solvers who are incompatible will not solve the problems within the time limits. Solvers which heuristically perform subsumption will fall in between, to show the level of compatibility.

A number of variables is chosen uniformly from 512-1536, and a length uniformly from 8-12. A clause assigning a unique solution to every variable is chosen. A bit is flipped, and it heads a contradiction with a single (uniformly random) variable assignment is added. Each clause is split by adding either the positive or negative form of a (uniformly randomly chosen) variable. The process is repeated until every clause is of the target length.

Clauses produced this way always have another (probably unique) clause with which their resolution can subsume both; until it is a univariable assignment or the lone contradiction.

Verifying Hash Table Safety Properties in AWS C99 Package with CBMC

Muhammad Osama and Anton Wijs
Department of Mathematics and Computer Science
Eindhoven University of Technology, Eindhoven, The Netherlands
{o.m.m.muhammad, a.j.wijs}@tue.nl

***Abstract*—In this paper, state-of-the-art proofs are generated with harness using the CBMC bounded model checker for the Amazon Web Services C99 core package. In this submission, we check the safety properties of the *Hash Table find* routine with various *loop unwinding* settings as opposed to the *String compare* submitted last year. The generated proof has proven to be reasonably hard to solve using modern SAT solvers. It has many variable-clause redundancies which are not only challenging for a SAT solver but also useful to assess the performance of different simplification techniques.**

I. Introduction

Bounded Model Checking (BMC) [1]–[3] determines whether a model M satisfies a certain property φ expressed in temporal logic, by translating the model checking problem to a propositional satisfiability (SAT) problem or a Satisfiability Modulo Theories (SMT) problem. The term *bounded* refers to the fact that the BMC procedure searches for a counterexample to the property, i.e., an execution trace, which is bounded in length by an integer k. If no counterexample up to this length exists, k can be increased and BMC can be applied again. This process can continue until a counterexample has been found, a user-defined threshold has been reached, or it can be concluded (via k-induction [2]) that increasing k further will not result in finding a counterexample. CBMC [4], [5] is an example of a successful BMC model checker that uses SAT solving. CBMC can check ANSI-C programs. The verification is performed by *unwinding* the loops in the program under verification a finite number of times, and checking whether the bounded executions of the program satisfy a particular safety property [6]. These properties may address common program errors, such as null-pointer exceptions and array out-of-bound accesses, and user-provided assertions.

II. Benchmarks

In this paper, we are interested in verifying the safety properties of the *find* routine implemented in the Hash Table data structure of the Amazon Web Services (AWS) C99 core package. The proof covers the following:

- Memory allocation failure and access violations
- Pointer/floating-point overflow
- Data types conversion

We generated 21 different formulas using different hash table sizes in the range $[10, 30]$, with an incremental step. These sizes chosen carefully to produce SAT formulas with 100% coverage of all functionalities. All problems are written in this format:

```
hash_table_find_safety_size_<x>
```

where `x` denotes the size value. The first and the last formulas are solved via MiniSat [7] within 209 and 2738 seconds respectively on a machine with Intel Xeon Platinum 8260 processor operating at base clock of 2.4 GHz. The solving time of the rest of the benchmarks are expected to fall within that range.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, “Symbolic Model Checking without BDDs,” in *Proc. of TACAS (Mar. 1999), Amsterdam, The Netherlands*, ser. LNCS, vol. 1579. Springer, 1999, pp. 193–207.
[2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, “Bounded model checking,” *Advances in Computers*, vol. 58, pp. 117–148, 2003.
[3] M. Osama and A. Wijs, “GPU Acceleration of Bounded Model Checking with ParaFROST,” in *Proc. of CAV (Jul. 2021), USA*, ser. LNCS, vol. 12760. Springer, 2021, pp. 447–460.
[4] E. M. Clarke, D. Kroening, and F. Lerda, “A Tool for Checking ANSI-C Programs,” in *Proc. of TACAS (Mar. 2004), Barcelona, Spain*, ser. LNCS, vol. 2988. Springer, 2004, pp. 168–176.
[5] D. Kroening and M. Tautschnig, “CBMC - C Bounded Model Checker - (Competition Contribution),” in *Proc. of TACAS (Apr. 2014), Grenoble, France*, ser. LNCS, vol. 8413. Springer, 2014, pp. 389–391.
[6] D. Kroening and O. Strichman, *Decision Procedures - An Algorithmic Point of View, Second Edition*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.
[7] N. Eén and N. Sörensson, “An Extensible SAT-solver,” in *Proc. of SAT (May 2003) Santa Margherita Ligure, Italy*, ser. LNCS, vol. 2919. Springer, 2003, pp. 502–518.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Pigeon Hole and Mutilated Chessboard with Mixed Constraint Encodings and Symmetry-Breaking

Cayden R. Codel, Joseph E. Reeves, Randal E. Bryant
Carnegie Mellon University, Pittsburgh, United States

INTRO

The pigeonhole and mutilated chessboard problems are challenging benchmarks for most SAT solvers not employing special reasoning techniques. The solvers that do employ special techniques can efficiently solve the canonical versions of these two problems, but may fail with even slight problem variations. In a previous competition, we submitted formulas from a benchmark family of perfect matching problems on bipartite graphs that generalizes the pigeonhole and mutilated chessboard problems [1]. These formulas were generated from moderately dense bipartite graphs. For this competition we present additional formulas drawn from the bipartite graphs representing the pigeon hole problem, and the mutilated chessboard problem. These problems provide more structure for solvers to make use of, but variation of encodings may deter less robust solving techniques. We add varying amounts of symmetry-breaking clauses to formulas, which should improve a general SAT solver's performance but may worsen the performance of a solver with special solving techniques.

PIGEON HOLE AND MUTILATED CHESSBOARD PROBLEMS

Random bipartite graphs are used to explore non-structured problem instances of the perfect matching problem. Our benchmark generator can create encodings of perfect matching problems on random bipartite graphs for any n and m. In this description, we look at structured instances of the bipartite matching problem, namely the pigeon hole problem and the mutilated chessboard problem.

The pigeon hole problem involves placing $n+1$ pigeons into n holes, where no hole can contain two pigeons. This problem is encoded using at-most-one (AMO) constraints stating each hole contains at most one pigeon, and at-least-one (ALO) constraints stating each pigeon is in at least one hole. This problem along with the mutilated chessboard problem have exponentially sized resolution proofs [2], [3].

The mutilated chessboard problem involves covering an $n \times n$ chessboard that has two opposite corners removed with 2×1 dominoes. This formula is encoded using AMO constraints stating at most one domino is placed on each tile, and ALO constraints stating at least one domino is placed on each tile. Viewing the problem as a bipartite graph, this simply adds more edges to the graph, where each edge is a possible domino placement, and vertices in the graph are tiles on the board.

AMO ENCODINGS

There are several ways to encode an AMO constraint into conjunctive normal form (CNF). Some encodings introduce new variables (auxiliary variables) to the formula. These variables can be important for improving solver performance on many problems.

We present three AMO encodings: Pairwise, Sinz, and Linear,

Pairwise$(x_1, ..., x_n)$ is the pairwise set of binary clauses with no auxiliary variables:

$(\overline{x}_i \vee \overline{x}_j)$ with $1 \leq i < j \leq n$

Sinz$(x_1, ..., x_n)$ introduces signal variables that propagate the AMO condition:

$\overline{x}_i \vee s_i$ for $1 \leq i \leq n$ $\qquad$ $\overline{s}_i \vee s_{i+1}$, $\overline{s}_i \vee \overline{x}_{i+1}$ for $1 \leq i < n$

Linear$(x_1, ..., x_n)$ introduces variables to split up the Pairwise encoding when $n > 4$:

$$\text{Pairwise}(x_1, x_2, x_3, y) \wedge \text{AMO}(\overline{y}, x_4, .., x_n)$$

The **Mixed** AMO constraint option selects one of the three AMO encodings at random for each AMO independently.

SYMMETRY BREAKING CLAUSES

The problems listed above have many natural symmetries. For example, pigeon 1 can be placed in hole 1 or hole 2, and pigeon 2 can be placed in hole 1 or hole 2. We can break the symmetry by saying pigeon 1 cannot be placed in hole 1 if pigeon 2 is in hole 2. This can be represented as a binary clause and added to the formula. Some proof systems allow the addition of such clauses. When generalizing this to the problem of a perfect matching on a bipartite graph, the symmetry-breaking clauses can be added to disallow all but one of the possible perfect matchings on any $K_{2,2}$ or 6-cycle subgraph. For the pigeon hole problem, we can take pigeons as the vertices in one partition and holes as the vertices in the other. The example for pigeon $1, 2$ and hole $1, 2$ would represent a $K_{2,2}$ in the graph, and the symmetry can be broken with a single clause.

Our generator can be configured to randomly add a specified percent of symmetry-breaking clauses to the formula. We first count the total number of possible symmetry-breaking clauses, then randomly add the clauses until the percentage is met.

Benchmarks

We submitted 20 benchmarks to the 2023 SAT Competition. Eight formulas are the pigeon hole problem $n = 15...18$ with mixed constraint encodings, and either 15% or 35% of the symmetry-breaking clauses added. Twelve formulas are the mutilated chessboard $n = 16, 18, 20, 22$ with 25%, 35%, or 45% of the symmetry breaking clauses added. All formulas are unsatisfiable. The tool can be found at https://github.com/jreeves3/BiPartGen-Artifact.

References

[1] C. Codel, J. Reeves, M. Heule, and R. Bryant, "Bipartite perfect matching benchmarks," in *Proceedings of Pragmatics of (SAT)*, 2021.

[2] M. Alekhnovich, "Mutilated chessboard problem is exponentially hard for resolution," *Theoretical Computer Science*, vol. 310, no. 1, pp. 513–525, 2004.

[3] A. Urquhart, "Hard examples for resolution," *J.ACM*, vol. 34, no. 1, pp. 209–219, 1987.

TABLE I: Problem sizes at which timeout occurred on symmetry-broken pigeonhole (top four rows) and mutilated chessboard problems (bottom four rows). The columns indicate the fraction of symmetry-breaking clauses added. If a solver didn't time out on instances of size $n \leq 24$ for pigeonhole and $n \leq 30$ for mutilated chessboard, then runtime (in seconds) is reported in parens.

Encoding	Direct			Sinz			Linear		
Solver	0%	50%	100%	0%	50%	100%	0%	50%	100%
PGBDD	11	11	23	10	8	22	14	10	23
KISSAT	12	(246.87)	(0.03)	14	(3.93)	(0.04)	13	(3.12)	(0.048)
LINGELING	(0.01)	(0.03)	(0.08)	15	(4.75)	(0.20)	(0.01)	(0.01)	(0.04)
SADICAL	(3.00)	14	20	9	13	(10.11)	9	14	23
PGBDD	18	16	14	18	8	8	–	–	–
KISSAT	20	22	26	14	22	26	–	–	–
LINGELING	(0.02)	(0.02)	(0.03)	18	20	24	–	–	–
SADICAL	14	16	16	14	14	16	–	–	–

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Simplified and Randomized Formula REG^N

Shuolin Li[1], Chu-Min Li[12], Mao Luo[3], Jordi Coll[4], Mohamed Sami Cherif[1], Djamal Habet[1] and Felip Manyà[4]
[1]*Aix Marseille Univ, Université de Toulon*
CNRS, LIS, Marseille, France
shuolin.li, mohamedsami.cherif, djamal.habet@lis-lab.fr

[2]*Université de Picardie Jules Verne*
Amiens, France
chu-min.li@u-picardie.fr

[3]*School of Computer Science, Hubei University of Technology*
Wuhan, China
luomao@hbut.edu.cn

[4]*Artificial Intelligence Research Institute*
CSIC, Bellaterra, Spain
jcoll, felip@iiia.csic.es

***Abstract*—This document is an introduction of the benchmarks submitted to the SAT Competition 2023. These benchmarks are based on the REG^N formulas described in [1], some simplifications and randomizations are done to make the benchmark size and hardness reasonable.**

Preliminary

In [1], Goerdt uses two integers M and N such that $N = 2^M$, to generate a variable matrix Var^N, the variables in the matrix following the natural order. He divides each row into *sections*, a section at row i consists of 2^i adjacent variables. Then, he equally divides each section into two *halfsections*. The notation *Sec* x (*Halfsec* x) to show the unique section (halfsection) containing variable x. in addition, Row x (Col x) is the unique row (column) containing x. For $X, Y \subseteq$ Var^N, X is *covered* by Y if and only if $X \subseteq \cup_{y \in Y}$Col y.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24

Fig. 1. An example of matrix Var^8

For example, given $M = 3$, then $N = 8$, Goerdt gives the variable matrix Var^8 in Fig 1 in [1]. There are 4 sections in the first row, separated by short vertical bars, 2 sections in the second row and 1 in the third.

Formula MPHP^N

Goerdt then defines in [1] formulas MPHP^N (Modified PigeonHole Principle) by modifying the formulas used by Haken in [2], which encode the pigeonhole principle. Each formula MPHP^N consists of positive and negative clauses.

The disjunction of all variable in the same column of the matrix Var^N forms a positive clause. So, a MPHP^N formula has N positive clauses.

The negative clauses are binary as $\{\neg x, \neg y\}$, x and y belong to the same section, but x is chosen from the left half section and y from the right half section.

The formulas MPHP^N are UNSAT. Goerdt shows that there exists a polynomial regular resolution proof for them [1]. Since general resolution is stronger than regular resolution, there is a polynomial general resolution proof of it, too. By making some modifications to the formulas MPHP^N, Goerdt gets formula REG^N, which can be proved to have a polynomial general resolution proof but no polynomial regular resolution proof [1].

Formula REG^N

To define formula REG^N, Goerdt first introduces a few new concepts. Given an integer K, let $M = 3K$ and $N = 2^{3K}$ be the number of rows and columns in matrix Var^N defined in preliminary. Goerdt defines *Third 1* to be the set of variables from the first K rows, *Third 2* from the second K rows and *Third 3* from the last K rows. In Fig 1, $K = 1, M = 3, N = 8$, Third 1 to Third 3 respectively contain the first, second and third row.

Then, Goerdt uses *Cor13* to denote the correspondence relation between Third 1 and Third 3. For $x \in$ Row i ($i \in [1, K]$), $y \in$ Row $2K + i$, $y \in$ Cor13(x) if and only if x is covered by Sec y, but not covered by Halfsec y. *Cor32* denotes the correspondence relation between Third 3 and Third 2. For $x \in$ Row $2K + i$, $y \in$ Row $K + i$, $y \in$ Cor32(x) if and only if Sec y covers x, but Halfsec y does not cover x.

A formula REG^N consists of positive and negative clauses as a formula MPHP^N. The positive clauses in REG^N are the same as in MPHP^N and the negative clauses can be divided into three subsets Third 1, Third 2 and Third 3.

The negative clauses for Third 1 can be divided into two types. The first is a group of ternary clauses, which can be showed in the form $\{\neg x, \neg y, \neg z\}$. x and y come from the same section at Row i ($i \in [1, K]$) in Third 1, but from different halfsections, x belongs to the left half and y to the right half, and $z \in$ Cor13(x) (Note that Cor13(x) = Cor13(y)). The second is the set of clause $\{\neg x, \neg y\} \cup$ Sec z, the definition of x, y are the same as in the first type.

The negative clauses for Third 2 are simply binary clauses $\{\neg x, \neg y\}$, in which x, y belong to the same section from Row $K + 1$ to $2K$ in Third 2, x coming from the left half section and y from the right.

There are four kinds of negative clauses in Third 3. In $\{\neg x, \neg y, \neg z, \neg w\}$, x and y come from the left and the right half of the same section from Row $2K + 1$ to $3K$, respectively. And $z \in \mathrm{Cor32}(x)$, $w \in \mathrm{Cor32}(y)$. The other three sets of clauses are $\{\neg x, \neg y, \neg z\} \cup \mathrm{Cor32}(y)$, $z \in \mathrm{Cor32}(x)$; $\{\neg x, \neg y, \neg w\} \cup \mathrm{Cor32}(x)$, $w \in \mathrm{Cor32}(y)$; $\{\neg x, \neg y\} \cup \mathrm{Cor32}(x) \cup \mathrm{Cor32}(y)$, the definition of x, y are the same as before.

The formulas REG^N are unsatisfiable and have polynomial general resolution proof, but only have superpolynomial regular resolution proof [1]. Goerdt uses these formulas to say that general resolution is exponentially stronger than regular resolution.

Simplified and Randomized Formula REG^N

With the growth of K, the REG^N formula grows very rapidly, resulting in large CNF benchmark which are too hard to solve in practice. In order to obtain a group of benchmarks with reasonable size and hardness, we modify the formula REG^N.

We keep all positive clauses as before. For negative clauses, we keep clauses for Third 1, and remove all clauses for Third 3. After removing Third 3 clauses, the number of rows in Third 2 won't affect other clauses, so we introduce a new variable $L, L \in [0, K]$ for the number of rows in Third 2. Now the simplified Var^N has $2K + L$ rows, K rows in Third 1, L rows in Third 2 and K rows in Third 3. The simplified REG^N formula is made of the same positive clauses as before, the negative clauses in Third 1 and the negative clauses in new and smaller Third 2.

Note that this simplification is due to the need to generate suitable benchmarks in practice, it may have changed the theoretical property of the original REG^N formula, so that an optimal variable ordering may lead to a polynomial regular resolution proof for the changed formulas. However, we still hope these changed formulas can distinguish regular resolution and general resolution in practice, because a SAT solver usually does not miss that optimal ordering for regular resolution. Furthermore, the bounded variable elimination in the CDCL solvers is based on regular resolution, while CDCL is equivalent to general resolution under a few assumptions. We hope that these changed formulas can be used to guide the improvement of bounded variable elimination implementation and its cooperation with CDCL.

The variables in the original Var^N matrix follows the natural order. We introduce a random parameter $RandomSeed$ to shuffle the order of variable distributions. The randomization won't change the benchmarks' size but it may affect the hardness because it changes the initial solving path of the SAT solver. These random orderings might be used to illustrate the robustness of the variable ordering in variable elimination.

The name of the benchmarks we submit is "REGRandom-Kx-Ly-Seed$z$.cnf", in which x is the number of K, y is the number of rows in Third 2 and z is RandomSeed.

References

[1] A. Goerdt, "Regular Resolution Versus Unrestricted Resolution," SIAM J. Comput., vol. 22, no. 4, pp. 661–683, Aug. 1993, doi: 10.1137/0222044.

[2] A. Haken, "The intractability of resolution," Theoretical Computer Science, vol. 39, pp. 297–308, Jan. 1985, doi: 10.1016/0304-3975(85)90144-6.

Logical Equivalence Checking of Arithmetic Benchmarks

Zhihui Xie[‡], Xu Liu[‡], Wanqian Luo[†], Junhua Huang[†], Hui-Ling Zhen[†], Xijun Li[†], Mingxuan Yuan[†] and Shuai Li[‡]
[†]Huawei Noah's Ark Lab
[‡]Shanghai Jiao Tong University
{luowanqian1, huangjunhua15, zhenhuiling2, xijun.li, Yuan.Mingxuan}@huawei.com
{fffffarmer, liu_skywalker, shuaili8}@sjtu.edu.cn

***Abstract*—We describe SAT encoding for logical equivalence checking of arithmetic benchmarks**

I. Introduction

Logical equivalence checking (LEC) plays an important role in EDA. Its application is to verify functional equivalence of combinational circuits after multi-level logic synthesis. In a typical scenario, there are two structurally different implementations of the same design, and the problem is to prove their functional equivalence.

In a LEC flow, the two circuits are transformed into a single circuit called *miter* [1] derived by combining the pairs of inputs with the same names and feeding the pairs of outputs with the same names into XOR gates. The miter is a combinational circuit with the same inputs as the original circuit and the is constant 0 if and only if the two original circuits produce identical output values under all possible input assignments

II. Encoding

The data for constructing the miter circuits are obtained from the publicly available dataset EPFL benchmark [2], which consists of three subsets, namely arithmetic, random/control and MtM circuits. Here we use the arithmetic dataset to generate the benchmark to be submitted.

First we combine the two circuits into a miter circuit using XOR, and then we transform the circuit into a CNF using Tseytin Encoding [3]. To increase the difficulty of solving the generated CNF, we transform the formula. The transformation is mainly the replacement of variables into two new variables, which are combined by specific operators. Here we use CNFgen [4] to perform an OR or XOR transformation of the CNF.

On the generated CNF, we also perform a series of random shuffling operations to further increase the computing difficulty, which include variable permutation, clause permutation and polarity flip. In detail, the variable permutation changes the order of variables within each clause, the clause permutation changes the order of clauses within the CNF formula, and the polarity flip will flip the phase of literals randomly.

References

[1] D. Brand, "Verification of large synthesized designs," Proc. ICCAD 1993, pp. 534 -537.
[2] Amarú L, Gaillardon P E, De Micheli G, "The EPFL combinational benchmark suite," Proceedings of the 24th International Workshop on Logic Synthesis (IWLS), 2015.
[3] https://en.wikipedia.org/wiki/Tseytin_transformation
[4] https://github.com/MassimoLauria/cnfgen

CNF Generation of Arithmetic Circuits

Zhihan Chen, Yuhang Qian, Xindi Zhang, Shaowei Cai*

State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China
{chenzh,zhangxd,caisw}@ios.ac.cn,i@yuhangq.com

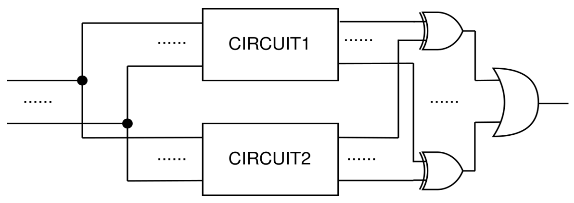


Fig. 1. Miter Circuit

Introduction

The difficulty in logical equivalence checking (LEC) of arithmetic circuits has been a challenging problem for many years. In recent times, the SAT-SWEEPING framework [2] has emerged as a widely used approach for equivalence verification due to its ability to provide an efficient solution to the problem. However, its solving performance is highly dependent on the sat solvers that are used. Therefore, the performance of SAT solvers in arithmetic circuit verification is worthy of attention. This document submits some instances of miter circuit of array and Wallace tree multipliers for SAT Competition 2023.

Miter Circuit

Miter circuit [1] is a logic circuit that compares the outputs of the two circuits being verified. Fig. 1 is an example. Specifically, the construction of the miter circuit involves connecting the inputs of two circuits whose equivalence is to be checked, with both circuits sharing the same input. The corresponding outputs are then connected via XOR gates, and all of these XOR gates are taken as inputs to an OR gate, with the output of the OR gate serving as the output of the miter circuit. If there is a set of inputs for which the output of the miter circuit is 1, it indicates that the circuits are not equivalent. Conversely, if such inputs do not exist, the two circuits are equivalent.

Generation of Benchmarks

Firstly, implement a basic circuit generator to generate array multipliers and Wallace tree multipliers of different bit widths. The circuit is generated in AIG (And-Inverter Graph) format, which includes only AND and NOT logic. Then, build a miter circuit using the two different implementation multiplier circuits, which also output in AIG format. Finally, convert the AIG format to CNF format.

Converting AIG format to CNF format is very easy. Only the encoding of AND gates needs to be considered, for example $c = a \ And \ b$ can be encoded to CNF $(\neg c \vee a) \wedge (\neg c \vee b) \wedge (c \vee \neg b \vee \neg a)$ As for the NOT logic in AIG, it can be achieved by inverting the symbol.

Benchmarks

The equivalence checking of multipliers has the property of small scale but high computational difficulty. So we generated multipliers of 10 to 16 bits and evaluated the difficulty of each example. To further simplify the problem, these instances only need to verify the equivalence of one of the outputs of multiplier circuit. Then we selected some examples with moderate difficulty for submission. The benchmarks are shown in Table I.

TABLE I
MULTIPLIER EQUIVALENCE CHECKING BENCHMARK

benchmark	num_vars	num_clauses
multiplier_13bits__miter_14.cnf	3728	11082
multiplier_13bits__miter_15.cnf	3728	11082
multiplier_13bits__miter_16.cnf	3728	11082
multiplier_13bits__miter_17.cnf	3728	11082
multiplier_14bits__miter_14.cnf	4376	13018
multiplier_14bits__miter_15.cnf	4376	13018
multiplier_14bits__miter_16.cnf	4376	13018
multiplier_14bits__miter_17.cnf	4376	13018
multiplier_14bits__miter_18.cnf	4376	13018
multiplier_14bits__miter_19.cnf	4376	13018
multiplier_15bits__miter_18.cnf	5052	15038
multiplier_15bits__miter_19.cnf	5052	15038
multiplier_15bits__miter_20.cnf	5052	15038
multiplier_15bits__miter_23.cnf	5052	15038
multiplier_15bits__miter_24.cnf	5052	15038
multiplier_16bits__miter_19.cnf	5776	17202
multiplier_16bits__miter_20.cnf	5776	17202
multiplier_16bits__miter_21.cnf	5776	17202
multiplier_16bits__miter_22.cnf	5776	17202
multiplier_16bits__miter_27.cnf	5776	17202

References

[1] D. Brand. Verification of large synthesized designs. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 534–537. IEEE, 1993.

[2] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een. Improvements to combinational equivalence checking. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 836–843, 2006.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Encoding Reduced Simon Cipher

Zhongyi Zhang

[1]State Key Laboratory of Information Security, Institute of Information Engineering,Chinese Academy of Sciences, Beijing, China
zhangzhongyi0714@iie.ac.cn

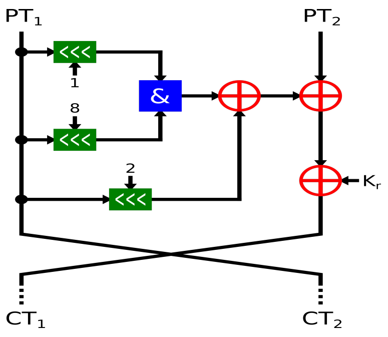


Fig. 1. One round of the Simon cipher

***Abstract*—The propositional satisfiability problem (SAT) has a wide range of applications in cryptography. This document describes several CNF instances encoded from a simplified Simon cipher, which are submitted to SAT Competition 2023.**

I. The Simon Cipher and CNF Encoding

The Simon cipher [1] is a famous block cipher, which is in the form of the balanced Feistel structure. The studies of the Simon cipher started in 2011, and the Simon cipher is released by the NSA in 2013. The goal of Simon cipher is to work effectively on a wide range of Internet of Things devices while retaining an acceptable security level .

Since the Simon cipher is a balanced Feistel cipher with n-bit word, the block length is $2n$. The key length is a m multiple of n, where $m = 2, 3$, or 4. Therefore, a Simon cipher instance can be denoted as Simon$2n/nm$. The operations of the Simon cipher round function are shown in Fig. 1. Tabel I shows the combinations of block sizes, key sizes, and the number of rounds, which are suggested by Simon [1].

The Simon Cipher can be seen as a special circuit with only bit-wise AND gates and XOR gates. Noting that the shifting operations in the Simon cipher can be viewed as EQL gates, which can be eliminated in the encoding process. And the gates can be easily encoded into CNF by the Tseitin encoding methods [2]. According to the Tseitin encoding methods, the 2 inputs AND gate $a = b$ AND c can be encoded to $\neg a \vee b \vee c$, $\neg b \vee a$, and $\neg c \vee a$; the 2 inputs XOR gate $a = b$ XOR c can be encoded to $a \vee \neg b \vee \neg c$, $\neg a \vee b \vee \neg c$, $\neg a \vee \neg b \vee c$ and $a \vee b \vee c$.

However, even with the easiest Simon cipher structure [1], where the block size is 32 bits and the number of rounds is 32, state-of-the-art sequential SAT solvers are still unable to solve the encoded CNFs. For the convenience of evaluation in SAT Competition, we reduce the structure of the Simon cipher so that it can be solved by some of the popular SAT solvers in a reasonable time. The methods are as follows:

- We remove the key scheduling methods, which means that the keys are set the same for each round.
- The number of keys is reduced from 64 to 32 by randomly generating 32 EQL relationships among keys.
- The number of rounds is reduced from 32 to at least 16.

TABLE I
The parameters supported by standard Simon cipher

Block Size (bits)	32	48	48	64	64	96	96	128	128	128
Key Size (bits)	64	72	96	96	128	96	144	128	192	256
Rounds	32	36	36	42	44	52	54	68	69	72

II. Benchmark Selection

The submitted benchmarks are shown in Table II, where the number of rounds (R), the number of variables (V), and the number of clauses (C) are given. For each round, we randomly create three instances with three different seeds.

TABLE II
Details of the selected instances

Benchmark Names	R	V	C
16_0.cnf, 16_1.cnf, 16_2.cnf	16	2688	8896
17_0.cnf, 17_1.cnf, 17_2.cnf	17	2848	9440
18_0.cnf, 18_1.cnf, 18_2.cnf	18	3008	9984
19_0.cnf, 19_1.cnf, 19_2.cnf	19	3168	10528
20_0.cnf, 20_1.cnf, 20_2.cnf	20	3328	11072
21_0.cnf, 21_1.cnf, 21_2.cnf	21	3488	11616
22_0.cnf, 22_1.cnf, 22_2.cnf	22	3648	12160
23_0.cnf, 23_1.cnf, 23_2.cnf	23	3808	12704
24_0.cnf, 24_1.cnf, 24_2.cnf	24	3968	13248
25_0.cnf, 25_1.cnf, 25_2.cnf	25	4128	13792
26_0.cnf, 26_1.cnf, 26_2.cnf	26	4288	14336
27_0.cnf, 27_1.cnf, 27_2.cnf	27	4448	14880
28_0.cnf, 28_1.cnf, 28_2.cnf	28	4608	15424
29_0.cnf, 29_1.cnf, 29_2.cnf	29	4768	15968
30_0.cnf, 30_1.cnf, 30_2.cnf	30	4928	16512
31_0.cnf, 31_1.cnf, 31_2.cnf	31	5088	17056
32_0.cnf, 32_1.cnf, 32_2.cnf	32	5248	17600

References

[1] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The simon and speck families of lightweight block ciphers. *cryptology eprint archive*, 2013.

[2] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pages 466–483, 1983.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

SAT Instances based on the Set Covering Problem with Conflict

Jiongzhi Zheng[1,2] Mingming Jin[1,2] Kun He[1,2] Zhuo Chen[1,2] Jinghui Xue[1,2]
[1]School of Computer Science and Technology, Huazhong University of Science and Technology, China
[2]Hopcroft Center on Computing Science, Huazhong University of Science and Technology, China
{jzzheng,mingmingk,brooklet60,ciaozer,jh_xue}@hust.edu.cn

I. Introduction

We propose a new variant of the well-known set covering problem [1], called set covering problem with conflict (WSCPC). In the last competition, we generate 20 UNSAT instances by transforming 20 WSCPC instances. This year we generate 20 SAT instances by transforming 4 WSCPC instances.

II. The Weighted Set Covering Problem with Conflict

Given a set of items $S = \{s_1, ..., s_m\}$ and a set of elements $E = \{e_1, ..., e_n\}$, where each item covers a subset of E and each element e_j ($j \in \{1, ..., n\}$ has a positive weight w_j, a conflict graph $G = (V, E')$ where the node set V consists of all the items in S, an edge (s_i, s_j) belonging to E' indicates that items s_i and s_j are conflict with each other. The goal of WSCPC is to find a subset $S' \subset S$ that any two items in S' are not conflict, and the total weight of the covered elements is maximized.

III. Transforming WSCPC into SAT

Firstly, a Weighted Partial MaxSAT [2], [3] instance can be generated by a WSCPC instance as follows. We use each item s_i to represent a variable v_i, each element e_j that covered by k items $\{s_{j1}, ..., s_{jk}\}$ to represent a soft clause with weight w_j that consists of the positive literal of variables $\{v_{j1}, ..., v_{jk}\}$. We further use each pair of conflict items s_a and s_b to generate a hard clause, which consists of the negative literal of v_a and v_b.

The Weighted Partial MaxSAT instance generated by a WSCPC instance can be easily transformed to an SAT instance by selecting all the hard clauses and $p\%$ random soft clauses to obtain a new CNF formula.

IV. Benchmarks

We first generate four SCPC instances with 700, 800, 900, and 1000 items, respectively. For each instance, the number of elements is 5 times the number of items, the density of the conflict graph is 0.1, and each element is covered by 55 randomly selected items.

Each SAT instance is named SCPC-m-p, where m is the number of items in the corresponding SCPC instance, and p is the percent of selecting soft clauses.

All the instances are SAT.

References

[1] E. Balas, M. W. Padberg, "On the set-covering problem," Oper. Res., vol. 20(6), pp. 1152-1161, 1972.

[2] J. Zheng, K. He, J. Zhou, Y. Jin, C. M. Li, F. Manyà, "BandMaxSAT: A Local Search MaxSAT Solver with Multi-armed Bandit," IJCAI 2022.

[3] J. Zheng, K. He, J. Zhou, "Farsighted Probabilistic Sampling based Local Search for (Weighted) Partial MaxSAT," AAAI 2023.

PROOF CHECKER DESCRIPTIONS

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

GRAT: a formally verified (UN)SAT proof checker

Proof Checker Proposal

Peter Lammich
FMT Group, EEMCS department
University of Twente
Enschede, The Netherlands
p.lammich@utwente.nl 0000-0003-3576-0504

***Abstract*—We propose the GRAT proof checker toolchain as a verified proof checker suitable for SAT competitions. It accepts proofs in the DRAT format, and is verified down to a functional implementation in Standard ML. On benchmarks drawn from recent SAT competitions, it's performance is similar to that of drat-trim.**

I. Introduction

The GRAT toolchain accepts DRAT (ASCII and binary format) as input. The result is formally verified with Isabelle/HOL, down to the integer sequence representing the formula. The trusted code base of the verification is Isabelle/HOL's kernel and code generator, compilation and running of the extracted Standard ML code with MLton, and a thin command line wrapper and formula parser written in Standard ML.

Our tool chain follows a two step approach, with a highly optimized but unverified first step, and a formally verified second step. As the first step only acts as certificate preprocessor, it is not part of the trusted code base.

On a set of benchmarks drawn from the 2016 and 2017 SAT competitions, our full toolchain performed faster than the unverified (then state-of-the-art) tool *drat-trim*. We have confirmed that our tool is still usable for modern SAT competitions, by testing it on benchmarks from the 2022 SAT competition.

A detailed description can be found in [2], [3] and [4]. Here, we briefly summarize the main aspects, and report on the new set of benchmarks.

GRAT's webpage is https://www21.in.tum.de/~lammich/grat/, and the project is maintained as part of the IsaFOL repository https://bitbucket.org/isafol/isafol/src/master/GRAT/. Download and build instructions are on the webpage.

II. Proof Format

Our toolchain supports the de-facto standard DRAT-format as input [5].

This is then processed by the unverified *gratgen* tool, which produces a certificate enriched with unit propagation information, in the GRAT-format. The GRAT certificate and the original formula is then passed to the verified *gratchk* tool, which either confirms unsatisfiability of the formula by printing the status line *s VERIFIED UNSAT*, or yields an error.

In the following, we sketch the GRAT-format.

Each clause is identified by a unique positive ID. The clauses of the original formula implicitly get the IDs $1 \dots N$. The lemma IDs explicitly occur in the certificate.

For memory efficiency reasons, we store the certificate in two parts: The lemma file contains the lemmas, and is stored in DIMACS format. During certificate checking, this part is entirely loaded into memory. The proof file contains the hints and instructions for the certificate checker. It is not completely loaded into memory but only streamed during checking.

The proof file is a binary file, containing a sequence (stored in reverse order) of 32 bit signed integers in 2's complement little endian format. The sequence is interpreted according to the following grammar:

```
proof      ::= rat-counts item* conflict
literal    ::= int32 != 0
id         ::= int32 > 0
count      ::= int32 > 0
rat-counts ::= 6 (literal count)* 0
item       ::= uprop | del | rup-lem | rat-lem
uprop      ::= 1 id* 0
del        ::= 2 id* 0
rup-lem    ::= 3 id id* 0 id
rat-lem    ::= 4 literal id id* 0 cand-prf* 0
cand-prf   ::= id id* 0 id
conflict   ::= 5 id
```

The checker maintains a *clause map* that maps IDs to clauses, and a *partial assignment* that maps variables to true, false, or undecided. Partial assignments are extended to literals in the natural way. Initially, the clause map contains the clauses of the original formula, and the partial assignment maps all variables to undecided. Then, the checker iterates over the items of the proof, processing each item as follows:

- `rat-counts` This item contains a list of pairs of literals and the count how often they are used in RAT proofs. This map allows the checker to maintain lists of RAT candidates for the relevant literals, instead of gathering the possible RAT candidates by iterating over the whole clause database for each RAT proof, which is expensive. Literals that are not used in RAT proofs at all do not occur in the list. This item is the first item of the proof.
- `uprop` For each listed clause ID, the corresponding clause is checked to be unit, and the unit literal is assigned to true. Here, a clause is unit if the unit literal is undecided, and all other literals are assigned to false.
- `del` The specified IDs are removed from the clause map.

- `rup-lem` The item specifies the ID for the new lemma, which is the next unprocessed lemma from the lemma file, a list of unit clause IDs, and a conflict clause ID. First, the literals of the lemma are assigned to false. The lemma must not be blocked, i.e. none of its literals may be already assigned to true[1]. Note that assigning the literals of a clause C to false is equivalent to adding the conjunct $\neg C$ to the formula. Second, the unit clauses are checked and the corresponding unit literals are assigned to true. Third, it is checked that the conflict clause ID actually identifies a conflict clause, i.e. that all its literals are assigned to false. Finally, the lemma is added to the clause-map and the assignment is rolled back to the state before checking of the item started.
- `rat-lemma` The item specifies a pivot literal l, an ID for the lemma, an initial list of unit clause IDs, and a list of candidate proofs. First, as for *rup-lemma*, the literals of the lemma are assigned to false and the initial unit propagations are performed. Second, it is checked that the provided RAT candidates are exhaustive, and then the corresponding *cand-prf* items are processed: A *cand-prf* item consists of the ID of the candidate clause D, a list of unit clause IDs, and a conflict clause ID. To check a candidate proof, the literals of $D \setminus \{\neg l\}$ are assigned to false, the listed unit propagations are performed, and the conflict clause is checked to be actually conflict. Afterwards, the assignment is rolled back to the state before checking the candidate proof. Third, when all candidate proofs have been checked, the lemma is added to the clause map and the assignment is rolled back.
  To simplify certificate generation in backward mode, we allow candidate proofs referring to arbitrary, even invalid, clause IDs. Those proofs must be ignored by the checker.
- `conflict` This is the last item of the certificate. It specifies the ID of the conflict clause found by unit propagation after adding the last lemma of the certificate (*root conflict*). It is checked that the ID actually refers to a conflict clause.

III. Evaluation

A. Usage Example

We give a simple example on how to use our toolchain:

To verify that a formula stored in the DIMACS file `unsat.cnf` is unsatisfiable, proceed as follows:

```
# Create a (binary) drat-file
> kissat -q unsat.cnf unsat.drat
s UNSATISFIABLE
# Process into proof (gratp) and lemmas (gratl) file
> gratgen unsat.cnf unsat.drat \
   -o unsat.gratp -l unsat.gratl -b
s VERIFIED
# Check against original formula
> gratchk unsat unsat.{cnf,gratl,gratp}
s VERIFIED UNSAT
```

[1]Blocked lemmas are useless for unsat proofs, such that there is no point to include them in the certificate.

To verify that a formula stored in the DIMACS file `sat.cnf` is satisfiable, proceed as follows:

```
# Produce variable assignment,
# as 0-terminated list of literals
> kissat -q sat.cnf | grep "^v" \
   | sed -re 's/^v//g' > sat.vars
# Check against original formula
> gratchk sat sat.{cnf,vars}
s VERIFIED SAT
```

B. MLtons Memory Manager

When running gratchk on machines with a lot of memory, we ran into two problems with MLtons default memory manager: First it will take half of the machine's memory before even starting to garbage collect. And, second, when it garbage collects, it will try to keep allocated 8 times the live memory size. Both behaviours are problematic: small problems will consume huge amounts of memory, making it impossible to verify many small problems in parallel on the same machine. Also, most of the memory that gratchk consumes is the storage for the formula and lemmas. Once the checking starts, only little additional memory is needed. However, MLtons memory manager will try to allocate 8 times the live size, which includes the (potentially large) formula and lemmas. In practice, this led to gratchk processes being killed by the out-of-memory killer.

While there is no ideal solution currently supported by MLton, we decided to apply a simple heuristic and limit the memory available to gratchk to 10 times the formula and lemma file size, and a minimum of 1GiB. In practice, this can be achieved by system tools, or by a runtime option to MLton, e.g.:

```
> gratchk @MLton max-heap 2G -- \
   unsat unsat.{cnf,gratl,gratp}
```

C. Theoretical Complexity

Our toolchain has polynomial complexity in the size of the input (drat) certificate and formula. While we have not estimated the precise complexities, we give a rough argument that the complexity is polynomial.

The first phase, *gratgen*, iterates over each clause in the certificate, and puts it into a two-watched-literals (twl) data structure. This clearly takes polynomial time. It then iterates backwards over the clauses. For each clause, the (inverted) literals are added as units, and then unit-propagation is performed. This also takes polynomial time. In case of a RAT clause, further clauses are gathered from the available clauses, and for each of those, another unit propagation is done (again, polynomial unit propagation for linearly many clauses). After checking each clause, the twl data structure is reverted to the state before that clause (which also takes polynomial time).

The second phase, *gratchk*, repeats the actions from the first phase, but iterating in a forwards fashion, and using extra information for unit propagation. Thus, it is also clearly polynomial.

D. Empirical Evaluation

We have extensively benchmarked our toolchain in [4], where we also compared it against the then-current versions of *drat-trim* and LRAT [1].

Our tool has not significantly changed since then, and we refer the reader to [4] for those results.

To check if our tool is still usable, we have run it on problems from the 2022 SAT competition's main track. We considered the winning solver Kissat_MABHyWalk, and the highest ranked non-Kissat based solver SeqFROST-ERE-All. We ran the solvers on all unsatisfiable problems they could solve in the competition to regenerate the certificates, and then used GRAT to verify the results. We benchmarked two configurations for gratgen: single-threaded and 8 parallel threads. Previous experiments have shown that more than 8 threads do not bring significant speedup.

1) Verified Problems: First of all, we could verify all 146 problems for Kissat and all 138 problems for SeqFROST. The single-threaded gratgen timed out on one Kissat problem, though.

2) Solving vs. Verification time: We compare the solving time with the verification time. Let t_s be the solving time, and t_v be the verification time, we compute, for each problem the ratio $r = t_v/t_s$, and then count for what percentage of the problems this ratio is less than .5, 1, 2, and 4. This is a sensible measure, as we expect the verification time to be related to the difficulty of the problem, and thus the solving time. Also, it estimates the extra time required to get a verified result. The result is displayed in the following table:

	$< .5$	< 1	< 2	< 4	#problems
Kissat-j8	70.5	85.6	93.8	97.3	146
SeqFROST-j8	76.8	89.1	96.4	99.3	138
Kissat-j1	26.9	60.0	87.6	93.8	145
SeqFROST-j1	24.6	50.7	81.9	97.1	138

That is, with 8 threads, we can verify more than 80% of the problems when allowing the same time for verification as for solving. In single-threaded mode, it's still more than half of the problems. And more than 90% of the problems will be solved when allowing a factor of 2 (8 threads) or 4 (1 thread), respectively.

3) Drat Certificate vs Grat Certificate Size: Next, we compare the size of the drat certificate produced by the SAT solver to the size of the enriched (grat) certificate produced by the first phase of our tool. This is of concern as the certificates have to be stored on disk, and thus, should not be excessively big. As for the time, we determine the ratio grat-size over drat-size, and count the percentage of problems below certain ratios.

	$< .5$	< 1	< 2	< 4	#problems
Kissat	46.6	54.1	84.9	97.9	146
SeqFROST	50.0	54.3	81.9	97.8	138

We observe that the generated grat certificate is smaller than the original drat certificate in more than half of the cases, and rarely exceeds factor 4. This is due to the trimming heuristics in gratgen, which, similar to drat-trim, tries hard to eliminate as many useless lemmas as possible. In many cases, this elimination removes more than the extra unit-propagation information that is added by gratgen.

IV. Formal Verification

The crucial part of our toolchain is the *gratchk* tool, which takes as input the original formula and a certificate in GRAT format, and then verifies that the formula is actually unsatisfiable. It also supports a mode for verifying satisfiable formulas, which takes a list of true literals as proof.

The *gratchk* tool is written in Standard ML, and compiled using the MLton compiler. The top-level is an unverified command line interface, which interprets the commands, and parses the specified files into an array of integers. The array contains a representation of the formula, followed by a representation of the proof. It then calls the core functions *verify_sat_impl* and *verify_unsat_split_impl*, which are exported from an Isabelle formalization using Isabelle's code generator.

```
val verify_sat_impl
  : int array → nat → unit → (_, _) sum
val verify_unsat_split_impl
  : int array → ('a → int * 'a)
    → nat → nat → nat * 'a → unit → (_, _) sum
```

For these functions, we have proved the following lemmas in Isabelle:

```
theorem verify_sat_impl_correct:
  <DBi ↦a DB>
    verify_sat_impl DBi F_end
  <λresult. DBi ↦a DB
    * ↑(¬isl result ⟹ verify_sat_spec DB F_end)>

theorem verify_unsat_impl_correct:
  <DBi ↦a DB>
    verify_unsat_split_impl DBi prfn F_end it prf
  <λresult. DBi ↦a DB
    * ↑(¬isl result ⟹ verify_unsat_spec DB F_end)>
```

The preconditions of these Hoare triples state that the argument *DBi* points to an array holding the elements *DB*. This array is not changed by the functions (it occurs unchanged in the postcondition), and these Hoare-triples imply termination of the program, as well as that it does not change any memory apart from what it allocates itself.

The original formula is stored in *DB[1..<F_end]*. (*DB[0]* is used as a guard by our implementation). The result of the functions are from an exception monad, represented by a sum type. The second parts of the postconditions state that, if no exception is raised, the formula stored at *DB[1..<F_end]* is satisfiable or unsatisfiable respectively. In case of the unsat proof, the other parameters *prfn, it, prf* are used to represent the proof, but they have no influence on the statement of this lemma: regardless of their values, an accepted formula is always unsat (If we pass nonsense, however, we will likely get an exception).

To express when a formula is (un)sat, we have two (proved equivalent) specifications. The first version relies on a function F_α that maps lists of integers to our internal representation of SAT formulas, and the predicate *sat* that specifies if a formula is satisfiable:

```
definition verify_sat_spec DB F_end ≡
    1 ≤ F_end ∧ F_end ≤ length DB
  ∧ (let lst = tl (take F_end DB) in
        F_invar lst ∧ sat (F_α lst))

definition verify_unsat_spec DB F_end ≡
    1 ≤ F_end ∧ F_end ≤ length DB
  ∧ (let lst = tl (take F_end DB) in
        F_invar lst ∧ ¬sat (F_α lst))
```

These specifications state that *F_end* is in range, and that *DB*[*1..<F_end*] (in Isabelle: *tl (take F_end DB)*) is a valid (*F_invar*) representation of a satisfiable or unsatisfiable, respectively, formula.

To increase the trust in these specifications, we prove them equivalent to a version that only relies on basic list operations: First, we use the function *tokenize :: int list ⇒ int list list*, which splits a list into its zero-terminated components. To sanity-check this function, we prove that, for a list that ends with a zero (i.e., contains no open clause at the end), its result is the unique inverse of concatenation:

```
definition concat0 ll = concat (map (λl . l@[0]) ll)
lemma unique_tokenization:
  assumes l≠[] ⟹ last l = 0
  shows ∃₁ls. (0∉⋃set (map set ls) ∧ concat0 ls = l)
    and tokenize l = (THE ls.
          0∉⋃set (map set ls) ∧ concat0 ls = l)
```

where *THE* is the definite description operator.

Next, we define an assignment from integers to Booleans to be consistent iff a negative value is mapped to the opposite of its absolute value:

```
definition assn_consistent :: (int ⇒ bool) ⇒ bool
  where assn_consistent σ
    = (∀x. x≠0 ⟹ ¬ σ (-x) = σ x)
```

Finally, we characterize an (un)satisfiable input by the (non)existence of a consistent assignment that assigns at least one literal of each clause to true. Thus, we prove the following alternative characterizations of our specifications:

```
lemma verify_sat_spec DB F_end = (
  1≤F_end ∧ F_end ≤ length DB ∧ (
  let lst = tl (take F_end DB) in
    (lst≠[] ⟹ last lst = 0)
  ∧ (∃σ. assn_consistent σ
        ∧ (∀C∈set (tokenize 0 lst). ∃l∈set C. σ l))))

lemma verify_unsat_spec DB F_end = (
  1 < F_end ∧ F_end ≤ length DB ∧ (
  let lst = tl (take F_end DB) in
      last lst = 0
  ∧ (∄σ. assn_consistent σ
        ∧ (∀C∈set (tokenize 0 lst). ∃l∈set C. σ l))))
```

In the case of unsatisfiability, the bounds have been adjusted to exclude the empty formula, which is trivially satisfiable.

A. Trusted Code Base

Our approach relies on the correctness of the following components

- Isabelle/HOL's inference kernel.
- Isabelle/HOL's code generator to Standard ML.
- The Imperative/HOL extension of the code generator.
- The correct formalization of what a Hoare-triple means.
- The correct specification of the correctness properties.
- The command line interface and DIMACs file parser.
- The correctness of the ML compiler and execution environment.

Where possible, we have tried to keep these trusted components as simple as possible. For example, we have proved two equivalent forms of the correctness specification, and limited the unverified parser to parse the DIMACs file into an array of integers. The interpretation of these integers as list of clauses is done inside Isabelle.

References

[1] M. Heule, W. Hunt, M. Kaufmann, and N. Wetzler. Efficient, verified checking of propositional proofs. In *Proc. of ITP*. Springer, 2017.

[2] P. Lammich. Efficient verified (UN)SAT certificate checking. In *Proc. of CADE*. Springer, 2017.

[3] P. Lammich. The GRAT tool chain - efficient (UN)SAT certificate checking with formal correctness guarantees. In *SAT*, pages 457–463, 2017.

[4] P. Lammich. Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.*, 64(3):513–532, 2020.

[5] N. Wetzler, M. J. H. Heule, and W. A. Hunt. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In *Proc. of SAT 2014*, pages 422–429. Springer, 2014.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

VERIPB and CAKEPB in the SAT Competition 2023

Bart Bogaerts Ciaran McCreesh Magnus O. Myreen Jakob Nordström Andy Oertel Yong Kiam Tan

I. SUMMARY

The pseudo-Boolean proof format used for the proof checker VERIPB [1] supports proof logging for decision, enumeration, and optimization problems, as well as problem reformulations, all in a unified format. So far, VERIPB has been used for proof logging of enhanced SAT solving techniques [2], [3], pseudo-Boolean CDCL-based solving [4], constraint programming [5], [6], subgraph solving [7], [8], and MaxSAT solving [9], [10], and this list of applications is expected to keep growing. This description briefly summarizes how a restricted version of the proof format can be used to certify unsatisfiability of CNF formulas in the SAT competition 2023. A complete documentation of the proof format can be found at https://gitlab.com/MIAOresearch/software/VeriPB/-/blob/satcomp2023_checker/satcomp23/documentation_SAT_competition_2023.pdf.

II. QUICKSTART GUIDE FOR BOOLEAN SATISFIABILITY (SAT) PROOF LOGGING

This section contains the bare minimum of information needed to use VERIPB and CAKEPB as proof checkers for Boolean satisfiability (SAT) solvers with pseudo-Boolean proof logging. A good way to learn more (in addition to reading this document) might be to study the example files in the directory `tests/integration_tests/correct/` in the repository [1] and run VERIPB with the options `--trace --useColor`, which will output detailed information about the proofs and the proof checking.

A. Running the Proof Checkers

If a SAT solver with pseudo-Boolean proof logging has solved the instance `input.cnf`, the generated proof `input.pbp` can be checked by VERIPB and CAKEPB by runnning the following commands:

```
# Translate to kernel format proof
veripb --cnf --proofOutput translated.pbp \
  input.cnf input.pbp

# Check the kernel proof
cake_pb_cnf input.cnf translated.pbp
```

The first command recompiles the pseudo-Boolean proof `input.pbp` into a more restricted "kernel-format" proof `translated.pbp` using VERIPB, after which the kernel proof is checked using CAKEPB. In case of successful recompilation, VERIPB will output:

```
# Running veripb as shown above
...
Verification succeeded
```

Upon successful proof checking, CAKEPB will report success on the standard output stream:

```
# Running cake_pb_cnf as shown above
s VERIFIED UNSAT
```

All errors are reported on standard error.

B. Proof Format

The syntactic format of a pseudo-Boolean proof of unsatisfiability for a CNF formula as expected by the version of VERIPB proposed for the SAT competition 2023 is

```
pseudo-Boolean proof version 2.0
f ⟨N⟩
Derivation section
output NONE
conclusion UNSAT : ⟨id⟩
end pseudo-Boolean proof
```

where $\langle N \rangle$ should be the number of clauses in the formula and *Derivation section* should contain the actual proof which derives contradiction as the pseudo-Boolean constraint with constraint ID $\langle id \rangle$.

In pseudo-Boolean format, a disjunctive clause like

$$x_1 \lor \overline{x}_2 \lor x_3 \tag{1a}$$

is represented as the inequality

$$x_1 + \overline{x}_2 + x_3 \geq 1 \tag{1b}$$

claiming that at least one of the literals in the clause is true (i.e., takes value 1), and this inequality is written as

```
+1 x1 +1 ~x2 +1 x3 >= 1 ;
```

in the OPB format [11] used by VERIPB. The proof checker can also read CNF formulas in the DIMACS and WDIMACS formats used for SAT solving and MaxSAT solving, respectively. For such files, VERIPB will parse a clause

```
1 -2 3 0
```

to be identical to (1b), and the variables should be referred to in the pseudo-Boolean proof file as `x1`, `x2`, `x3`, et cetera.

DRAT proofs can be transformed into valid VERIPB proofs by simple syntactic manipulations. Most of the proof resulting from a CDCL SAT solver is the ordered sequence of clauses learned during conflict analysis. Since all such clauses are guaranteed to be *reverse unit propagation (RUP)* clauses, the easiest way to provide pseudo-Boolean proof logging for a learned clause (1a) would be to write

```
rup +1 x1 +1 ~x2 +1 x3 >= 1 ;
```

in the derivation section of the pseudo-Boolean proof.

If instead the clause (1a) is a *resolution asymmetric tautology (RAT)* clause that is RAT on the literal x_1, then this is written as

```
red +1 x1 +1 ~x2 +1 x3 >= 1 ; x1 -> 1
```

in the pseudo-Boolean proof using the more general *redundance-based strengthening* rule. And if the RAT literal would instead have been $\overline{x}_2$, this would have been indicated by ending the proof line above by `x2 -> 0` instead.

Finally, in order to delete the clause (1a), the deletion command

```
del spec +1 x1 +1 ~x2 +1 x3 >= 1 ;
```

is issued. An important difference from DRAT proofs is that deletion is made also for unit clauses, i.e., clauses containing only a single literal—DRAT proof checkers typically ignore such deletion commands. Another crucial difference is that all clauses learned during CDCL execution need to be written down in the proof log, including unit clauses. If unit clauses are missing in a DRAT proof, the proof checkers will typically be helpful and silently infer and add the missing clauses. No such patching of formally incorrect proofs is offered by VERIPB.

It should be noted, though, that if all the reasoning performed by some particular SAT solver can efficiently be captured by standard DRAT proof logging, then there is no real reason to use pseudo-Boolean proof logging for that solver. Pseudo-Boolean proof logging becomes relevant only if the solver uses more advanced techniques such as, for instance, cardinality reasoning, Gaussian elimination, or symmetry breaking. We refer the reader to [2] and [3], respectively, for detailed descriptions of how to do efficient pseudo-Boolean proof logging for the latter two techniques. A detailed description of the cutting planes proof system and proof steps supported in the augmented for VERIPB and the kernel format for CAKEPB is available at https://gitlab.com/MIAOresearch/software/VeriPB/-/blob/satcomp2023_checker/satcomp23/documentation_SAT_competition_2023.pdf.

III. FORMALLY VERIFIED PROOF CHECKING

The kernel proof checker CAKEPB has been formally verified in the HOL4 theorem prover [12] using the CAKEML suite of tools for program verification, extraction, and compilation [13]–[15]. In this section, we present the verification guarantees for CAKEPB_CNF, a version of CAKEPB equipped with a DIMACS CNF parser frontend for UNSAT proof checking with pseudo-Boolean proof logging.

A. *Verified Correctness Theorem for* CAKEPB_CNF

The end-to-end verified correctness theorem for CAKEPB_CNF is shown in Figure 1. This theorem can be intuitively understood in four parts, corresponding to the indicated lines (2)–(5):

- The theorem assumes (2) that the CAKEML-compiled machine code for CAKEPB_CNF is executed in an x64 machine environment set up correctly for CAKEML. The definition of cake_pb_cnf_run is shown below, where the first line (wfcl cl ∧ wfFS fs ∧ ...) says the command line cl and filesystem fs match the assumptions of CAKEML's FFI model. The second line says that the compiled code (cake_pb_cnf_code) is correctly set up for execution on an x64 machine.

  cake_pb_cnf_run $cl\ fs\ mc\ ms \overset{\text{def}}{=}$
  wfcl cl ∧ wfFS fs ∧ STD_streams fs ∧ hasFreeFD fs ∧
  installed_x64 cake_pb_cnf_code $mc\ ms$

- Under these assumptions, the CAKEPB_CNF program is guaranteed to never crash (3). However, it may run out of resources such as heap or stack memory (extend_with_resource_limit ...). In these cases, CAKEPB_CNF will fail gracefully and report out-of-heap or out-of-stack on standard error.
- Upon termination, the CAKEPB_CNF program will output some (possibly empty) strings out and err to the standard output and standard error streams, respectively (4).
- The key verification guarantee (5) is that, whenever the string "s VERIFIED UNSAT" is printed to standard output, the input CNF file (first command line argument) parses in DIMACS format to a CNF which is unsatisfiable. No other output is possible on standard output; error strings are always printed to standard error.

Internally, CAKEPB_CNF transforms input CNF clauses (in DIMACS format) to normalized pseudo-Boolean constraints, as exemplified by (1a) and (1b). This transformation is formally verified to preserve satisfiability as part of the end-to-end correctness theorem shown in Figure 1. Note that the CAKEPB_CNF tool has an essentially identical correctness theorem to an existing verified Boolean unsatisfiability proof checking tool [16]. In fact, these tools share exactly the same definitions of DIMACS CNF parsing, Boolean satisfiability semantics, and all of the CAKEML's standard assumptions.

B. *Complexity and Empirical Evaluation*

All of the commands in the kernel format are designed to minimize the need to search over the entire constraint database. For example, each implicational and deletion proof step can be performed in linear time with respect to the size of that step.

The only proof steps that scale linearly with respect to the size of the constraint database are redundancy and dominance-based strengthening steps. For either of these steps, the proof checker potentially needs to loop over the entire constraint database to check all the necessary proof goals. However, the maximum size of the database is linear in the size of the input formula and the proof. Therefore, the overall complexity of the verified proof checker is polynomial in the size of the input formula and proof, as required.

Table I shows an empirical evaluation of the verified proof checking pipeline on a selected suite of example proofs, generated using BREAKID [17][1] and KISSAT [18][2] to solve

[1]https://bitbucket.org/krr/breakid/src/veriPB/
[2]https://gitlab.com/MIAOresearch/tools-and-utilities/kissat_fork

$$
\begin{aligned}
&\vdash \mathsf{cake\_pb\_cnf\_run}\ cl\ fs\ mc\ ms \Rightarrow && (2)\\
&\quad \left.\begin{aligned}
&\mathsf{machine\_sem}\ mc\ (\mathsf{basis\_ffi}\ cl\ fs)\ ms \subseteq\\
&\quad \mathsf{extend\_with\_resource\_limit}\\
&\qquad \{\ \mathsf{Terminate\ Success}\ (\mathsf{cake\_pb\_cnf\_io\_events}\ cl\ fs)\ \} \wedge
\end{aligned}\right\} && (3)\\
&\quad \left.\begin{aligned}
&\exists\, out\ err.\\
&\quad \mathsf{extract\_fs}\ fs\ (\mathsf{cake\_pb\_cnf\_io\_events}\ cl\ fs) =\\
&\qquad \mathsf{SOME}\ (\mathsf{add\_stdout}\ (\mathsf{add\_stderr}\ fs\ err)\ out) \wedge
\end{aligned}\right\} && (4)\\
&\quad \left.\begin{aligned}
&\mathsf{if}\ out = \text{«s VERIFIED UNSAT\n»}\ \mathsf{then}\\
&\quad \mathsf{LENGTH}\ cl = 3 \wedge \mathsf{inFS\_fname}\ fs\ (\mathsf{EL}\ 1\ cl) \wedge\\
&\quad \exists\, fml.\\
&\qquad \mathsf{parse\_dimacs}\ (\mathsf{all\_lines}\ fs\ (\mathsf{EL}\ 1\ cl)) = \mathsf{SOME}\ fml \wedge\\
&\qquad \mathsf{unsatisfiable}\ (\mathsf{interp}\ fml)\\
&\mathsf{else}\ out = \text{«»}
\end{aligned}\right\} && (5)
\end{aligned}
$$

Fig. 1: The end-to-end correctness theorem for the CakeML pseudo-Boolean proof checker with a CNF parser

TABLE I: Example timings for verified proof checking using VeriPB and CakePB_cnf. All times are in seconds.

Benchmark	VeriPB Time (s)	CakePB Time (s)
queen14_14.col.14.cnf	6.5	52.3
harder-php{...}.cnf	9.3	30.5
Pb-chnl15-16_c18.cnf	13	43.2
tseitin_n104_d3.cnf	4.2	3.9
rphp_p6_r28.cnf	123	68.2

SAT competition instances of the last years and theoretical instances.

Acknowledgements

We wish to acknowledge the monumental contributions of Stephan Gocht [19], without whom there would not have been any pseudo-Boolean proof checker.

References

[1] "VeriPB: Verifier for pseudo-Boolean proofs," https://gitlab.com/MIAOresearch/software/VeriPB.

[2] S. Gocht and J. Nordström, "Certifying parity reasoning efficiently using pseudo-Boolean proofs," in *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, Feb. 2021, pp. 3768–3777.

[3] B. Bogaerts, S. Gocht, C. McCreesh, and J. Nordström, "Certified symmetry and dominance breaking for combinatorial optimisation," in *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI '22)*, Feb. 2022, pp. 3698–3707.

[4] S. Gocht, R. Martins, J. Nordström, and A. Oertel, "Certified CNF translations for pseudo-Boolean solving," in *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 236, Aug. 2022, pp. 16:1–16:25.

[5] J. Elffers, S. Gocht, C. McCreesh, and J. Nordström, "Justifying all differences using pseudo-Boolean reasoning," in *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, Feb. 2020, pp. 1486–1494.

[6] S. Gocht, C. McCreesh, and J. Nordström, "An auditable constraint programming solver," in *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 235, Aug. 2022, pp. 25:1–25:18.

[7] ——, "Subgraph isomorphism meets cutting planes: Solving with certified solutions," in *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, Jul. 2020, pp. 1134–1140.

[8] S. Gocht, R. McBride, C. McCreesh, J. Nordström, P. Prosser, and J. Trimble, "Certifying solvers for clique and maximum common (connected) subgraph problems," in *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, ser. Lecture Notes in Computer Science, vol. 12333. Springer, Sep. 2020, pp. 338–357.

[9] D. Vandesande, W. De Wulf, and B. Bogaerts, "QMaxSATpb: A certified MaxSAT solver," in *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR '22)*, ser. Lecture Notes in Computer Science, vol. 13416. Springer, Sep. 2022, pp. 429–442.

[10] J. Berg, B. Bogaerts, J. Nordström, A. Oertel, and D. Vandesande, "Certified core-guided MaxSAT solving," in *Proceedings of CADE-29*, 2023, accepted for publication.

[11] O. Roussel and V. M. Manquinho, "Input/output format and solver requirements for the competitions of pseudo-Boolean solvers," Jan. 2016, revision 2324. Available at http://www.cril.univ-artois.fr/PB16/format.pdf.

[12] K. Slind and M. Norrish, "A brief overview of HOL4," in *TPHOLs*, ser. LNCS, O. A. Mohamed, C. A. Muñoz, and S. Tahar, Eds., vol. 5170. Springer, 2008, pp. 28–32.

[13] Y. K. Tan, M. O. Myreen, R. Kumar, A. C. J. Fox, S. Owens, and M. Norrish, "The verified CakeML compiler backend," *J. Funct. Program.*, vol. 29, p. e2, 2019.

[14] A. Guéneau, M. O. Myreen, R. Kumar, and M. Norrish, "Verified characteristic formulae for CakeML," in *ESOP*, ser. LNCS, H. Yang, Ed., vol. 10201. Springer, 2017, pp. 584–610.

[15] M. O. Myreen and S. Owens, "Proof-producing translation of higher-order logic into pure and stateful ML," *J. Funct. Program.*, vol. 24, no. 2-3, pp. 284–315, 2014.

[16] Y. K. Tan, M. J. H. Heule, and M. O. Myreen, "cake_lpr: Verified propagation redundancy checking in CakeML," in *TACAS*, ser. LNCS, J. F. Groote and K. G. Larsen, Eds., vol. 12652. Springer, 2021, pp. 223–241.

[17] "Breakid," https://bitbucket.org/krr/breakid.

[18] "Kissat SAT solver," http://fmv.jku.at/kissat/.

[19] S. Gocht, "Certifying correctness for combinatorial algorithms by using pseudo-Boolean reasoning," Ph.D. dissertation, Lund University, Lund, Sweden, Jun. 2022.

*Proceedings of SAT Competition 2023: Solver and Benchmark Descriptions*volume B-2023-1 of Department of Computer Science Series of Publications B, University of Helsinki 2023 .

Verified LRAT and LPR Proof Checking with `cake_lpr`

Yong Kiam Tan   Marijn J. H. Heule   Magnus O. Myreen

I. SUMMARY

We present the `cake_lpr` proof checker [1] which is capable of checking proofs in either Linear RAT (LRAT) or Linear PR (LPR) proof formats. The LPR format is a backwards-compatible extension of LRAT. The checker is formally verified using CakeML and the HOL4 theorem prover; its formal proof is discussed in [1] and briefly in Section III. The DRAT and DPR proof formats are supported using `DRAT-trim` and `DPR-trim` as preprocessing tools, respectively.
The verified proof checker is available at:
https://github.com/tanyongkiam/cake_lpr
The `DRAT-trim` and `DPR-trim` tools are available at:
https://github.com/marijnheule/drat-trim
https://github.com/marijnheule/dpr-trim

A. Example

An outline of an end-to-end LRAT proof checking run is as follows:

```
# Assume the problem is input.cnf in DIMACS

... run SAT solver on input.cnf ...
... generate proof file input.drat ...

# Run drat-trim on the DRAT proof and
# generate LRAT file

drat-trim input.cnf input.drat -L input.lrat

# Run cake_lpr on the resulting LRAT proof

cake_lpr input.cnf input.lrat
```

If the proof checks successfully, `cake_lpr` will print to standard output:

```
s VERIFIED UNSAT
```

All other error messages, such as proof checking error, parsing error, out-of-memory error, will be printed to `stderr`. Solvers capable of generating LRAT proofs directly can skip the use of `DRAT-trim`. End-to-end proof checking for LPR proofs can be done similarly, using `DPR-trim` as the preprocessor for DPR proofs. It is also possible to convert DPR proofs to DRAT, then use `DRAT-trim`, but this approach is **not recommended** as it is significantly slower than checking DPR (and LPR) proofs directly [1].

II. SUPPORTED PROOF FORMATS

Formal descriptions of all proof formats are available in the cited publications [1], [2] and online. We give brief descriptions of the formats with concrete examples.

A. DRAT and LRAT

The DRAT format consists of a list of clause addition or deletion steps, one per line. All lines are terminated by `0`. Each added clause must have RAT redundancy with respect to the current formula.

```
<CLAUSE> 0
d <CLAUSE> 0
```

Concrete example:

```
1 -2 3 0    # Add clause x_1,!x_2,x_3
d 1 2 -3 0  # Del clause x_1,x_2,!x_3
```

The `DRAT-trim` tool can be used as a preprocessor to automatically convert an input DRAT proof to LRAT format. The latter format extends DRAT with a notion of clause IDs and proof hints for each line. The input CNF is assumed to be given IDs in ascending order from 1 to n where n is the number of clauses in the file. Addition lines in LRAT have the following format, where `<ID>` is a positive integer, `<IDs>` is a list of `<ID>`, and `[...]*` denotes 0 or more repetitions of the enclosed block:

```
<ID> <CLAUSE> 0 <IDs> [-<ID> <IDs>]* 0
```

The first `<ID>` is the clause ID to be assigned to `<CLAUSE>`. If `<CLAUSE>` has RAT redundancy, then the first literal in the clause is the pivot literal. The first block of `<IDs>` lists unit propagation steps starting from the blocking assignment for `<CLAUSE>`. If `<CLAUSE>` has RAT redundancy, then this first block is followed by 0 or more `-<ID>` `<IDs>` blocks, where `-<ID>` refers to the `<ID>`-th clause in the RAT proof and the corresponding `<IDs>` indicate unit propagation steps for that clause.

Deletion steps are written with a list of clause IDs rather than clauses. All the clauses with IDs in `<IDs>` are deleted.

```
<ID> d <IDs> 0
```

Concrete example:

```
# Add clause x_1,!x_2,x_3 at clause ID 15
# with RAT on pivot !x_2
15 -2 1 3 0 4 13 7 10 8 -5 2 4 -10 3 5 0
# Del clause IDs 13 14 15
# (ID 16 in front of the line is ignored)
16 d 13 14 15 0
```

A complexity analysis for the LRAT proof format is given in [2, Theorem 2], where asymptotically (keeping all parameters constant except number n of steps in proofs), the complexity is reported as $O(n^2 \log n)$; `cake_lpr` slightly improves

$$
\begin{array}{ll}
\left.\begin{array}{l}\vdash \textsf{cake\_lpr\_run}\ cl\ fs\ mc\ ms \Rightarrow\end{array}\right\} & (1)\\
\left.\begin{array}{l}\quad \textsf{machine\_sem}\ mc\ (\textsf{basis\_ffi}\ cl\ fs)\ ms \subseteq \\ \quad\quad \textsf{extend\_with\_resource\_limit} \\ \quad\quad\quad \{\ \textsf{Terminate Success}\ (\textsf{cake\_lpr\_io\_events}\ cl\ fs)\ \}\ \wedge\end{array}\right\} & (2)\\
\left.\begin{array}{l}\quad \exists\, out\ err. \\ \quad\quad \textsf{extract\_fs}\ fs\ (\textsf{cake\_lpr\_io\_events}\ cl\ fs) = \\ \quad\quad\quad \textsf{Some}\ (\textsf{add\_stdout}\ (\textsf{add\_stderr}\ fs\ err)\ out)\ \wedge\end{array}\right\} & (3)\\
\left.\begin{array}{l}\quad\quad \textsf{if}\ \ldots \\ \quad\quad \textsf{else if length}\ cl = 3\ \textsf{then} \\ \quad\quad\quad \textsf{if}\ out = \text{«s VERIFIED UNSAT\textbackslash n»}\ \textsf{then} \\ \quad\quad\quad\quad \textsf{inFS\_fname}\ fs\ (\textsf{el}\ 1\ cl)\ \wedge \\ \quad\quad\quad\quad \exists\, fml. \\ \quad\quad\quad\quad\quad \textsf{parse\_dimacs}\ (\textsf{all\_lines}\ fs\ (\textsf{el}\ 1\ cl)) = \textsf{Some}\ fml\ \wedge \\ \quad\quad\quad\quad\quad \textsf{unsatisfiable}\ (\textsf{interp}\ fml) \\ \quad\quad\quad \textsf{else}\ out = \text{«»} \\ \quad\quad \textsf{else}\ \ldots\end{array}\right\} & (4)
\end{array}
$$

Fig. 1. The end-to-end correctness theorem for the CakeML LPR proof checker. (Some irrelevant cases are elided with . . . for brevity).

the asymptotic bound to $O(n^2)$ because it uses constant-time rather than logarithmic-time lookup data structures [1]. Empirically, we have observed that most proofs generated by solvers in past SAT competitions are dominated by simple (non-RAT) steps. In that case, one may expect near-linear scaling from `cake_lpr`.

B. *DPR and LPR*

The DPR format extends DRAT so that added clauses are *propagation redundant* with respect to the current formula. Here, `<WIT>` is a list of literals which must start with the first literal in `<CLAUSE>`. Note that this is syntactically backwards compatible with DRAT (when `<WIT>` is empty).

```
<CLAUSE> <WIT> 0
```

The `DPR-trim` tool can be used as a preprocessor to automatically convert an input DPR proof to LPR format. The latter format extends DPR with clause IDs and proof hints in the same way LRAT extends DRAT. The only syntactic addition is the optional `<WIT>` after `<CLAUSE>`.

```
<ID> <CLAUSE> <WIT> 0 <IDs> [-<ID> <IDs>]* 0
```

The proof checking procedure for LPR is backwards compatible with LRAT, using `<IDs>` and `[-<ID> <IDs>]*` as unit propagation hints for propagation redundancy [1]. Deletion lines are identical for LPR and LRAT.

Concrete example:

```
# Add clause x_1,!x_2,x_3 at clause ID 15
# with PR witness !x_2,x_5
15 -2 1 3 -2 5 0 4 13 7 10 8 -5 2 4 -10 3 5 0
# Del clause IDs 13 14 15
# (ID 16 in front of the line is ignored)
16 d 13 14 15 0
```

The proof checking procedure for LPR is essentially the same as LRAT, i.e., with $O(n^2)$ complexity (assuming all other parameters are held constant).

III. Proof Checker Verification

Our proof checker, `cake_lpr`, is formally verified down to the level of its x64 machine code implementation, which eliminates the possibility of bugs arising from, e.g., compiler errors, code extraction, or other, unverified additions to (verified) source code. This is achieved by compiling its formally verified CakeML source code implementation, with a formally verified compiler for CakeML [1].

The key correctness theorem is shown in Fig. 1. To informally summarize:

- Line (1) assumes that the `cake_lpr` binary is executed in an x64 machine environment set up according to the standard CakeML assumptions.
- Lines (2) guarantees that `cake_lpr` will terminate successfully (i.e., no out of bounds array accesses, etc.); it may run out of either heap or stack memory (resource limits).
- Lines (3) says that, according to the CakeML file system model, there will be some strings printed to standard output and standard error.
- Lines (4) says (among other things) that, IF the string “s VERIFIED UNSAT” is printed onto standard output, then the first command line argument corresponds to a file, which parsed in DIMACS format, to a formula that is unsatisfiable. The DIMACS parser is verified to be left inverse to the DIMACS printer.

References

[1] Y. K. Tan, M. J. H. Heule, and M. O. Myreen, “cake_lpr: Verified propagation redundancy checking in CakeML,” in *TACAS*, ser. LNCS, J. F. Groote and K. G. Larsen, Eds., vol. 12652. Springer, 2021, pp. 223–241.

[2] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt Jr., M. Kaufmann, and P. Schneider-Kamp, “Efficient certified RAT verification,” in *CADE*, ser. LNCS, L. de Moura, Ed., vol. 10395. Springer, 2017, pp. 220–236.

Solver Index

Benchmark Index

Proof checker Index

Author Index