



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 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, rephrasing, 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 many 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 extent 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. Jarvisalo, and M. Suda, Eds., vol. B-2022-1. University of Helsinki, 2022.