CaDiCaL at the SAT Race 2019

Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University Linz

Our SAT solver CaDiCaL provides a clean, documented, easy to understand and modify state-of-the-art solver, based on CDCL [1] with inprocessing [2]. Earlier versions participated in the SAT competition 2017 and 2018. Here we only describe differences to these versions [3], [4]. Even though CaDiCaL performed well on unsatisfiable instances in the SAT Competition 2018, the performance on satisfiable instances was behind the top solvers in that competition. Thus a large part of the changes made and described in this note are motivated by trying to improve CaDiCaL on satisfiable instances without loosing its good performance on unsatisfiable instances.

Separate Decision Queue

The earlier versions of CaDiCaL already partially followed the advice given by Chanseok Oh in [5] to interleave (what we call) *stable* search phases focusing on satisfiable instances with almost no restarts and (again in our terminology) *unstable* search phases with the usual frequent but limited restarts schedule. In our new version we use a reluctant doubling scheme with base conflict interval 1024 for the stable phase.

However, the results of [5] also suggest to use a smoother increase of scores for the stable phase using a separate decision queue. We have integrated this idea. It required to add the usual exponential VSIDS scoring mechanism using a binary heap as in MiniSAT [6]. Thus this new version relies on its previous VMTF queue [7] only for the unstable search phases and on the exponential VSIDS for the stable search phase.

The “default” configuration submitted to the competition alternates stable and unstable phases, while the “unsat” configuration remains in the unstable search phase and the “sat” configuration vice versa only in the “stable” phase.

Local Search

Our local search solver YalSAT [8] solved 48 instances in the main track of the SAT Competition 2018 from which 30 instances were not solved by CaDiCaL and even one not solved by any other solver. It further solved 36 instances faster than any other solver. This shows that it should be beneficial to add a local search component to CaDiCaL. We already had YalSAT hooked up to Lingeling, which was successfully used in Treengeling in parallel solver threads. However controlling the amount of time allocated to YalSAT is difficult. It also requires to copy all clauses.

Therefore we added a simple local search component to CaDiCaL. As YalSAT it is based on ideas developed in ProbSAT [9]. In contrast to YalSAT and ProbSAT, we watch one literal in each clause instead of using counters. The broken (unsatisfiable) clauses are kept on a stack and traversed completely during each step (flipping a literal).

Local search is called from the *rephase* procedure [3], [4] which is scheduled in regular intervals. It can also be executed as preprocessing step for an arbitrary number of rounds, which in essence turns the solver into a local search solver (disabled by default). As initial assignment for local search we use the same assignments that would be selected in the CDCL loop for decision variables (actually the target phases—see next section—are always preferred, even for local search during unstable phases). The best assignment (falsifying the smallest number of clauses) determined during each local search round is exported back to the CDCL loop as saved phases.

Target and Best Phases

Probably the most important new technique is the use of *target phases*, which can be seen as a generalization of phase saving [10]. This well-known technique saves the last value assigned to a variable (its saved phase) and uses it as assignment value if a variable is selected as decision.[1]

In addition to these saved phases our new approach now also maintains an array of target and another array of best phases. The idea is to maximize the size of the trail without conflicts. Thus during backtracking the prefix of the trail is determined which did not (yet) lead to a conflict previous propagations. The values of the literals on the prefix are then saved as new target phases if this prefix is larger than the previously saved one. In stable search phases these target phases are preferred over saved phases [10] for decisions.

During *rephasing* [3], [4] saved phases are reset as before, except, that beside the new local search rephasing discussed above we have further a new *best* rephasing, which sets saved phases to the values of the largest previously reached trail without conflict and then resets these best phases. By default best rephasing is only performed during stable search phases.

Lucky Phases

Occasionally applications produce trivial formulas in the sense that they can be satisfied by for instance assigning all variables to false. Some of them also made it into the competition and therefore we implemented in Lingeling [11]

Supported by FWF, NFN Grant S11408-N23 (RiSE).

[1]Unfortunately there are now two uses of the word “phase” here, one for stable and unstable search phases, as well as for the values assigned to variables. We hope it is clear from the context which of the two interpretation is meant whenever we use “phase”.

a "lucky phase" detector. This has been ported to CaDiCaL and extended to detect horn clause benchmarks, which can be satisfied by assigning in forward or backward order all variables to the same constant (interleaved with propagation). For instance satisfiable multiplier miters [12] comparing correct and buggy multipliers can be satisfied by this new lucky phase procedure instantly if the inputs either appear consecutively at the beginning or at the end of the variable range.

Improvements to Preprocessing

Since (bounded) variable elimination [13] remains the most important pre- and inprocessing technique, we tried to improve its effectiveness even further. First, if variable elimination completed, the bound on the number of allowed zero additional clauses (difference between non-tautological resolvents and clauses with a candidate variable) is increased (exponentially from the default zero to 1,2,4,8,16) and all variables are again considered as candidates for elimination attempts. We further perform variable elimination by substitution [13] if we are able to extract AND or XOR gates. We also added eager backward subsumption and strengthening after each successful variable elimination, in addition to our fast forward subsumption algorithm [3] which is continued to be applied to redundant clauses too. Last we added a resolution limit, to reduce the time spent in variable elimination for large but easy to solve formulas. In the same spirit we limit the number of subsumption checks during forward subsumption.

As in previous versions the solver triggers failed literal *probing* (including hyper binary resolution and equivalent literal substitutions) independently from both *subsumption* (on redundant and irredundant clauses followed by vivification) and variable *elimination* (elimination rounds are interleaved with subsumption and optionally, but disabled by default, with blocked and covered clause elimination). These preprocessors can also be called for multiple rounds initially. Using a conflict limit this allows the solver to be used as a CNF preprocessor (the extension stack needed for solution reconstruction can be extracted as well).

Chronological Backtracking

The winner Maple_LCM_Dist_ChronoBT [14] of the main track in the SAT Competition 2018 implemented a combination of chronological backtracking with CDCL [15]. We have ported this idea to CaDiCaL and as in the original work backtrack chronologically if backjumping would jump over more than 100 levels, but otherwise do not limit its application. We further combine it with the idea of reusing the trail [16]. More details will appear in [17].

Incremental Solving and Model Based Testing

Finally we added a new approach [18] to incremental SAT solving which does not require to freeze variables (as in MiniSAT and Lingeling) in order to be combined with inprocessing. To implement such a combination correctly requires sophisticated API testing and accordingly we implemented a tightly integrated model based tester called Mobical following the principles reported in [19].

License

The solver remains open source under the MIT License. New versions are available at http://fmv.jku.at/cadical and https://github.com/arminbiere/cadical.

References

[1] J. P. M. Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185, pp. 131–153.
[2] M. Järvisalo, M. Heule, and A. Biere, "Inprocessing rules," in *IJCAR*, ser. Lecture Notes in Computer Science, vol. 7364. Springer, 2012, pp. 355–370.
[3] A. Biere, "CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017," in *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, ser. Dept. of Comp. Science Series of Publications B, vol. B-2017-1. Univ. of Helsinki, 2017, pp. 14–15.
[4] ——, "CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018," in *Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions*, ser. Dept. of Comp. Science Series of Publications B, vol. B-2018-1. Univ. of Helsinki, 2018, pp. 13–14.
[5] C. Oh, "Between SAT and UNSAT: the fundamental difference in CDCL SAT," in *SAT*, ser. Lecture Notes in Computer Science, vol. 9340. Springer, 2015, pp. 307–323.
[6] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. Lecture Notes in Computer Science, vol. 2919. Springer, 2003, pp. 502–518.
[7] 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.
[8] A. Biere, "Yet another local search solver and Lingeling and friends entering the SAT Competition 2014," in *Proc. SAT Competition 2014, Solver and Benchmark Descriptions*, ser. Dept. of Comp. Science Series of Publications B, vol. B-2014-2. Univ. of Helsinki, 2014, pp. 39–40.
[9] A. Balint and U. Schöning, "Choosing probability distributions for stochastic local search and the role of make versus break," in *SAT*, ser. Lecture Notes in Comp. Science, vol. 7317. Springer, 2012, pp. 16–29.
[10] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *SAT*, ser. Lecture Notes in Computer Science, vol. 4501. Springer, 2007, pp. 294–299.
[11] " Lingeling and Friends Entering the SAT Race 2015," FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep., 2015.
[12] A. Biere, "Collection of Combinational Arithmetic Miters Submitted to the SAT Competition 2016," in *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, ser. Dept. of Computer Science Series of Publications B, vol. B-2016-1. Univ. of Helsinki, 2016, pp. 65–66.
[13] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *SAT*, ser. Lecture Notes in Computer Science, vol. 3569. Springer, 2005, pp. 61–75.
[14] A. Nadel and V. Ryvchin, "Maple_LCM_Dist_ChronoBT: Featuring chronological backtracking," in *Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions*, ser. Dept.Dept. Computer Science Series of Publications B, vol. B-2018-1. Univ. of Helsinki, 2018, p. 29.
[15] ——, "Chronological backtracking," in *SAT*, ser. Lecture Notes in Computer Science, vol. 10929. Springer, 2018, pp. 111–121.
[16] 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.
[17] S. Möhle and A. Biere, "Backing backtracking," 2019, submitted.
[18] K. Fazekas, A. Biere, and C. Scholl, "Incremental inprocessing in SAT solving," 2019, submitted.
[19] C. Artho, A. Biere, and M. Seidl, "Model-based testing for verification back-ends," in *TAP*, ser. Lecture Notes in Computer Science, vol. 7942. Springer, 2013, pp. 39–55.