CaDiCaL, Kissat, Paracooba Entering the SAT Competition 2021

Armin Biere Mathias Fleury Maximilian Heisinger
Institute for Formal Models and Verification
Johannes Kepler University Linz

***Abstract*—This system description describes updates to our sequential SAT solvers CaDiCaL and Kissat submitted to the main track as well as updates to our distributed cube-and-conquer solver Paracooba submitted to the cloud track.**

CaDiCaL 1.4.0

The competition organizers decided to use CaDiCaL as basis for a "hack track". Version 1.4.0 of CaDiCaL used in this track differs from the version submitted to the SAT Competition 2020 [1] as follows.

First, our version of "reason side bumping" [2] not only bumps literals in reason clauses of literals in the learned clause, but, depending on a run-time recursion depth parameter, also bumps reason literals of literals in reason clauses recursively. By default the recursion depth limit was 1, which lead to the same behaviour as [2]. Now, in "stable mode", focusing on satisfiable instances with few restarts and smoothed bumping [3], we have increased this recursion depth limit to 2.

Second, to compute several statistics, CaDiCaL uses exponential moving averages, particularly for controlling restarts [4], [5]. Initializing these averages is non-trivial and actually leads to biased estimates. For instance, without proper initialization, the slow moving average of the LBD (glucose levels) of learned clauses ramps up too slowly, trailing the fast moving average, which in turn triggers unmotivated restarts initially. We proposed a partial solution in [5] and implemented another improvement based on over-approximating the smoothing factor geometrically. With CaDiCaL 1.4.0 we adopted the method for initializing exponential moving averages proposed in the ADAM approach [6], which maintains and uses a correction factor to obtain an unbiased average.

Finally, and most importantly, the version of CaDiCaL submitted to the SAT Competition 2020 unfortunately failed to export the assignment found in local search minimizing the number of falsified clauses back to the CDCL loop as saved phases (due to a change in semantics of `copy_phases`), which in essence rendered the local search component completely useless. And indeed, our post-competition experiments showed, that this "heuristic bug" resulted in solving fewer instances during the competition. In version 1.4.0 saved phases are again explicitly overwritten at the end of local search with the minimum assignment found during local search.

Supported by Austrian Science Fund (FWF) projects W1255-N23 and S11408-N23, by the LIT AI and LIT Secure and Correct Systems Labs and the LIT project LOGTECHEDU all three funded by the State of Upper Austria.

CaDiCaL SC2021

On top of version 1.4.0 of CaDiCaL, as used in the "CaDiCaL hack track", we have added a light version of (what we call) "shrinking" by Feng & Bacchus presented at SAT 2020. Like the original version [7], our version [8] shortens learned clauses by calculating the unique implication point (UIP) on every level of the conflict clause, restricted to not adding literals on lower levels. Unlike Feng & Bacchus's version, our algorithm is efficient enough to be executed unconditionally and minimizes the clause at the same time. This technique is particularly effective when long clauses are learned as in the planning track of the SAT Competition 2020. For more details please refer to [8].

Kissat SC2021

We have also implemented "shrinking" [8] in Kissat, which beside reducing the length of learned clauses yields the same run-time improvements on instances from the planning track, without degrading performance on other instances, even though the percentage of time spent in conflict analysis (including clause minimization and shrinking) goes up.

The local search procedure either imports decision phases from the CDCL solver or in an alternating fashion uses previously best assignments computed by local search, similar in spirit to the "cache" component in YalSAT [9]. This allows the local search to continue where it left off with the best assignments it found earlier. In addition of using a fixed CB value of 2.0 we also allow to interpolate it based on average clause length and further use three variants of fixed clause weights. These "strategies" are changed at each call to the local search procedure (as during "restarts" in YalSAT).

Beside using the new method described above to initialize exponential moving averages the new version of Kissat

- reduced the number of rephasing methods by removing the random and the flipped rephasing,
- computes backbones on the binary implication graph,
- schedules variable elimination without a priority queue,
- bounds reason side variable bumping, and

uses its default non-compact memory configuration for the competition in order to allow the solver to go beyond 24 GB main memory (as the organizers now announced to use 128 GB main memory during the competition).

Finally, we use a new method for semantic gate extraction in bounded variable elimination of a fixed candidate variable x.

Let $F = F_x \wedge F_{\bar{x}} \wedge F'$, where F_ℓ is the CNF of clauses of F which contain literal ℓ, with $\ell \in \{x, \bar{x}\}$, and where $(F_x \wedge F_{\bar{x}})$ is called the *environment* of x. The remaining clauses of F without x nor $\bar{x}$ are collected in F'. Given CNFs H_x, $H_{\bar{x}}$ where clauses in H_ℓ all contain ℓ, we define the set of resolvents

$$H_x \otimes H_{\bar{x}} = \{(C \vee D) \mid (C \vee x) \in H_x,\ (D \vee \bar{x}) \in H_{\bar{x}} \text{ and } (C \vee D) \text{ not trivial}\}.$$

As usual we interpret a CNF also as a set of clauses. Our goal is to eliminate x through resolution, that is replacing F by $(F_x \otimes F_{\bar{x}}) \wedge F'$.

Already in [10], which introduced the SATELITE preprocessor, it was proposed to extract subsets of "gate clauses" from F_x and $F_{\bar{x}}$ which encode "circuit gates" with output x, also called definitions of x. Resolving these gate clauses against each other results in tautological (trivial) resolvents, and, in particular, this situation allows to ignore resolvents between non-gate clauses (since those are implied).

Finding such gate clauses was originally based on syntactic pattern matching, in essence inverting the Tseitin encoding. For details and a semantic variant inspired by BDD algorithms and implemented in Lingeling [11] see [12]. In KISSAT we follow more recent semantic approaches with applications in model counting [13] and QBF reasoning [14], which use a SAT solver as oracle to find gate clauses.

Let x be a candidate variable, which is tried to be eliminated without increasing the number of clauses much, and, for which all necessary resolvents have to be generated. If syntactic pattern matching for a Tseitin encoding of an AND, XOR or IF-THEN-ELSE gate with x as output fails, then our new version of KISSAT tries to extract gate clauses semantically by checking satisfiability of $(F_x|_{\bar{x}}) \wedge (F_{\bar{x}}|_x)$, i.e., the formula which is obtained by removing the occurrences of x in F_x and of $\bar{x}$ in $F_{\bar{x}}$ and then conjoining the result. If this formula is unsatisfiable we compute a clausal core which in turn can be mapped back to original gate clauses G_x and $G_{\bar{x}}$ in the environment (by adding back x resp. $\bar{x}$). Let H_ℓ be the remaining clauses of F_ℓ with $F_\ell = G_\ell \wedge H_\ell$. Then it turns out that $F_x \otimes F_{\bar{x}}$ can be reduced to $(G_x \otimes G_{\bar{x}}) \wedge (G_x \otimes H_{\bar{x}}) \wedge (G_{\bar{x}} \otimes H_x)$ and thus $(H_x \otimes H_{\bar{x}})$ can be omitted.[1] The effect is that fewer resolvents are generated and thus more variables can be eliminated.

To see that the last formula can be omitted assume that $A \wedge B$ is unsatisfiable and thus $\bar{A} \vee \bar{B}$ is valid. Therefore

$$(A \vee C) \wedge (B \vee D) \equiv (A \vee C) \wedge (B \vee D) \wedge (\bar{A} \vee \bar{B}) \Rightarrow (C \vee D)$$

using in essence two resolution steps for the implication. Setting $(A, B, C, D) = (G_x, G_{\bar{x}}, H_{\bar{x}}, H_x)$ shows the rest.

KITTEN

In order to check satisfiability and compute clausal cores of these co-factors of the environment of a variable we have implemented a simple sub-solver KITTEN with in-memory proof tracing and fast allocation and deallocation. If the conjunction of the co-factors of the environment are unsatisfiable we reduce through the API in KITTEN its formula to the clausal core, shuffle clauses and run KITTEN a second time which usually results in a smaller core and thus fewer gate clauses (increasing chances that the variable is eliminated).

If only one co-factor contains core clauses, we derive a unit clause instead. In this case the learned clauses in KITTEN are traversed to produce a DRAT proof trace sequence for this unit. This is one benefit of using a proof tracing sub-solver in contrast to the BDD inspired approach in Lingeling [11], which can not produce DRAT proofs easily. This KITTEN feature of extracting proofs in memory is also essential to produce proofs for "SAT sweeping" discussed next.

KISSAT SC2021 SWEEP

As in the SAT Competition 2020 we submitted the default configuration "KISSAT SC2021 DEFAULT" as well as "KISSAT SC2021 SAT". The latter uses target phases also during focused mode [15] and usually works better for satisfiable instances. Instead of submitting a configuration specialized for unsatisfiable instances to the SAT Competition 2021, we decided to submit some work in progress, which in principle we expect to also work better on unsatisfiable instances.

Using KITTEN as sub-solver we perform semantically complete "SAT sweeping" of small *extended environments* around each variable, which works as follows. For each candidate variable we allocate a fresh instance of KITTEN and traverse in breadth first search (BFS) the variable incidence graph (in which two variables share an edge if they occur in the same clause) and copy all clauses up to a certain "depth" limit away (the number of BFS generations) from the candidate variable. We start with the default depth limit of 2 and also limit the total number of copied clauses (1000) and variables (100).

After copying the environment clauses, we let KITTEN compute a satisfiable assignment of the extended environment. Note that an unsatisfiable environment actually results in the whole formula to be unsatisfiable. From this satisfying assignment we produce a candidate list of backbone variables and a partition of equivalent literal candidates. The accumulated time spent in KITTEN is further limited by "ticks" as in KISSAT [15].

Then for each backbone candidate, we assume its negation and call the sub-solver again. If the result is unsatisfiable we learn the unit (and optionally extract a DRAT proof trace). Otherwise we use the satisfying assignment provided by KITTEN to refine both the backbone candidate list and the partition of equivalent literal candidates. By randomizing and every third call flipping the saved phases before calling the sub-solver the number of necessary calls is reduced substantially.

In the last part we then try to prove for each pair of remaining equivalent literal pairs, whether they imply each other, through two sub-solver calls with corresponding assumptions. If both calls are unsatisfiable, the two literals are equivalent and are merged in a global union-find data structures (and again optionally a DRAT proof sequence is extracted). This union-find data structure is consulted during the copy phase to add additional clauses of equivalent literals (as well as the

[1] Resolvents among gate clauses are not necessarily tautological though.

equivalence). A failing satisfiable call refines the equivalent literal candidate partition and the process continues until no more equivalent literal candidates are left.

The advantage of this approach is that the effort is heavily bounded, i.e., propagation over a large number of variables is avoided and solving is completely decoupled from the main solver. This version of KISSAT is still considered work in progress and for instance lacks better (re)scheduling of candidate variables. For more details and references on "SAT sweeping" see [16], which tries to achieve the same effect, but uses global blocked clause decomposition. A more similar approach but without using a dedicated sub-solver was implemented in our discontinued SAT solver SPLATZ [17].

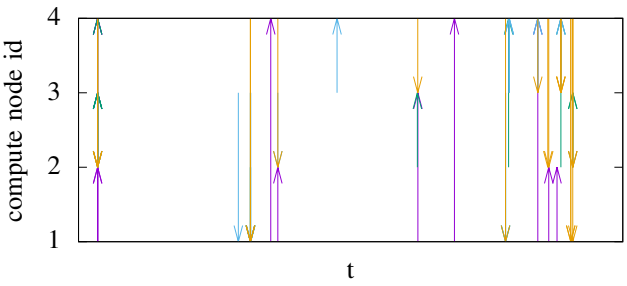
PARACOOBA SC2021

A new version of our solver PARACOOBA [15], [18] has been submitted to the cloud track. It is a distributed cube-and-conquer solver. The input DIMACS is analyzed and split once on the main node into multiple subproblem-branches. Each branch produces a tree of subproblems (cubes) that can be worked on independently. If branches of a cube-tree finish early, the branch is converted into a clause, which is distributed to all other solvers in the cluster. The first cube-tree is always favored when deciding on what task to work on next, so parallel cube trees do not starve the first tree-instance of available executors. Problems are re-split in case the CDCL-solver runs into a time-out depending on other solving times. Re-splits are done using the lookahead mechanism provided by CADICAL, which is also used as incremental solver to work on all generated cubes. To guard against problems that are faster to solve using a pure CDCL-based solving approach, KISSAT is running in parallel to the cube-and-conquer approach on the main node.

The architecture has been revised since last year, so that no ticks or delays are required anymore before tasks can be offloaded. Changes in a compute node's state are analyzed, compared to the last known received status of each known peer and sent only when the information gain of the new status is significant enough to warrant the transmission. This way, scheduling is more dynamic, relying less on overcommitting work (which lead to overfull queues) and instead offloading only when other nodes are nearly out of work. Furthermore, physical network connections no longer coincide with logical connections, which enables reconnects to work without losing previously sent messages. Timeouts are specified per peer and enforced with an improved keep-alive system that sends small messages if a connection has been idle for too long. This makes solving more resilient against network-based issues, even on commodity hardware. The latter also enables main nodes that only offload work and do not have workers themselves, which is useful for low-powered devices that only have wireless connections. Due to the low network bandwidth requirements, the connected peers can also be in the cloud and connected via SSH tunnels.

In order to test the improved offloading, the tracing tool DISTRAC was developed. This tool generates compact binary trace files that are concatenated after a run. A trace contains all required metadata (names, descriptions, causal dependencies between events) and can be analyzed using standard CLI tools, either in binary form, or after printing it in a textual column-based format. While analyzing the trace, events are sorted in order of system-wide occurrence, keeping causal relations intact. This enables easier debugging of network or solver events compared to merging log files, as filtering and selecting streams of data becomes easier.

One of the produced visualizations showing all offloads between compute nodes while solving a small benchmark can be seen below. Arrows are pointing from the compute node that currently works on a task to the new compute node that the task has been offloaded to. The Y-axis describes the compute node id, the X-axis the timestamp of an event. Highly utilized compute nodes offload their tasks to less utilized compute nodes, resulting in small clusters of arrows pointing to the same nodes, trying to maximize the active workers in the system without centralized coordination.



The solver has further been modularized into broker, communicator, solver, and runner components. These are either loaded at runtime as shared objects or statically linked into one binary. This mechanism enables using the PARACOOBA infrastructure for other problems, e.g., by implementing a custom solver module that uses the other mechanisms to automatically distribute tasks, or by changing the already provided solver to use different SAT-solvers. Automated unit- and system-tests work with dynamically loaded shared objects and can thus also be used to check third-party modules.

I. LICENSE

All our solvers are licensed under an MIT license and are available at https://github.com/arminbiere/cadical, http://fmv.jku.at/cadical for CADICAL, https://github.com/arminbiere/kissat, http://fmv.jku.at/kissat for KISSAT, and https://github.com/maximaximal/Paracooba for PARACOOBA.

REFERENCES

[1] A. Biere, "CaDiCaL at the SAT Race 2019," in *SAT Race 2019*, ser. Department of Computer Science Series of Publications B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2019-1. University of Helsinki, 2019, pp. 8–9.

[2] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for SAT solvers," in *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, N. Creignou and D. L. Berre, Eds., vol. 9710. Springer, 2016, pp. 123–140.

[3] C. Oh, "Between SAT and UNSAT: the fundamental difference in CDCL SAT," in *SAT 2015*, ser. LNCS, M. Heule and S. A. Weaver, Eds., vol. 9340. Springer, 2015, pp. 307–323. [Online]. Available: https://doi.org/10.1007/978-3-319-24318-4_23

[4] G. Audemard and L. Simon, "Refining restarts strategies for SAT and UNSAT," in *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, ser. Lecture Notes in Computer Science, M. Milano, Ed., vol. 7514. Springer, 2012, pp. 118–126.

[5] A. Biere and A. Fröhlich, "Evaluating CDCL restart schemes," in *Proceedings of Pragmatics of SAT 2015, Austin, Texas, USA, September 23, 2015 / Pragmatics of SAT 2018, Oxford, UK, July 7, 2018*, ser. EPiC Series in Computing, D. L. Berre and M. Järvisalo, Eds., vol. 59. EasyChair, 2018, pp. 1–17.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[7] N. Feng and F. Bacchus, "Clause size reduction with all-UIP learning," in *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, ser. Lecture Notes in Computer Science, L. Pulina and M. Seidl, Eds., vol. 12178. Springer, 2020, pp. 28–45.

[8] M. Fleury and A. Biere, "Efficient all-uip learned clause minimization," 2021, submitted.

[9] A. Biere, "Yet another local search solver and Lingeling and friends entering the SAT Competition 2014," in *Proc. of SAT Competition 2014 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, A. Balint, A. Belov, M. Heule, and M. Järvisalo, Eds., vol. B-2014-2. University of Helsinki, 2014, pp. 39–40.

[10] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 61–75.

[11] A. Biere, "Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010," Johannes Kepler University Linz, Tech. Rep., 2010.

[12] A. Biere, M. Järvisalo, and B. Kiesl, "Preprocessing in SAT solving," in *Handbook of Satisfiability*, 2nd ed., ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 391 – 435.

[13] J. Lagniez, E. Lonca, and P. Marquis, "Definability for model counting," *Artif. Intell.*, vol. 281, p. 103229, 2020.

[14] F. Slivovsky, "Interpolation-based semantic gate extraction and its applications to QBF preprocessing," in *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds., vol. 12224. Springer, 2020, pp. 508–528.

[15] 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.

[16] M. Heule and A. Biere, "Blocked clause decomposition," in *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, K. L. McMillan, A. Middeldorp, and A. Voronkov, Eds., vol. 8312. Springer, 2013, pp. 423–438.

[17] A. Biere, "Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016," in *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, and M. Järvisalo, Eds., vol. B-2016-1. University of Helsinki, 2016, pp. 44–45.

[18] M. Heisinger, M. Fleury, and A. Biere, "Distributed cube and conquer with paracooba," in *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, ser. Lecture Notes in Computer Science, L. Pulina and M. Seidl, Eds., vol. 12178. Springer, 2020, pp. 114–122.

[19] N. Feng and F. Bacchus, "Clause size reduction with all-UIP learning," in *SAT*, ser. LNCS, vol. 12178. Springer, 2020, pp. 28–45.

[20] R. Hickey, N. Feng, and F. Bacchus, "Cadical-trail, Cadical-alluip, Cadical-alluip-trail and Maple-LCM-Dist-alluip-trail at the SAT Competition," 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, p. 10.

[21] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatiorial design," *IEEE Trans. Software Eng.*, vol. 23, no. 7, pp. 437–444, 1997. [Online]. Available: https://doi.org/10.1109/32.605761

[22] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E. Choi, "Greedy combinatorial test case generation using unsatisfiable cores," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, D. Lo, S. Apel, and S. Khurshid, Eds. ACM, 2016, pp. 614–624. [Online]. Available: https://doi.org/10.1145/2970276.2970335