Abstract. The SAT solver CaDiCaL provides a rich feature set with a clean library interface. It has been adopted by many users, is well documented and easy to extend due to its effective testing and debugging infrastructure. In this tool paper we give a high-level introduction into the solver architecture and then go briefly over implemented techniques. We describe basic features and novel advanced usage scenarios. Experiments confirm that CaDiCaL despite this flexibility has state-of-the-art performance both in a stand-alone as well as incremental setting.

1 Introduction

Progress in SAT solving has a large impact on model checking, SMT, theorem proving, software- and hardware-verification, and automated reasoning in general, and, according to “The SAT Museum” [20], SAT solvers get faster and faster, at least on benchmarks consisting of a single formula. For incremental SAT solving it was less clear, particularly as preprocessing [24] and inprocessing [69] heavily contributing to this improvement were considered incompatible with incremental solving (the winners of the SAT competition main track rely on inprocessing since 2009 except in 2011/2012/2016 and since 2005 all on preprocessing).

A simple and elegant solution to this problem is due to the award winning incremental SAT solving approach [39] first implemented in CaDiCaL. It reverts clause removal, i.e., restores clauses removed during pre- and inprocessing, restrictively on a case-by-case basis. It allows incremental solving to make full use of pre- and inprocessing techniques, in contrast to less general solutions [87,89,90,112], without reducing their effectiveness nor burden the user to “freeze” and “melt” variables (“Don’t Touch” variables in [74]) as necessary with MiniSat [37].

This is the first tool paper on CaDiCaL, while previous, actually well cited, descriptions appeared only as system description in non-peer-reviewed SAT competition proceedings [14–16,18,21,22]. In general, even though “SAT is considered a killer app for the 21st century” (Donald Knuth), there are few tool papers on SAT solvers, with the prominent exception of MiniSat [37], which appeared in 2003 and was awarded the test-of-time award at SAT’22. The descriptions of...
CryptoMiniSat [106], Glucose [5] and InteLSAT [86] introduce the corresponding SAT solver and can be considered to be tool papers too though.

Development of CaDiCaL was triggered by discussions at the “Theoretical Foundations of SAT Solving Workshop” in 2016 at the Fields Institute in Toronto, where it became apparent that both theoreticians and practitioners in SAT have a hard time understanding how practical SAT solving evolved, what key components there are in modern SAT solvers and, most importantly, that it was apparently getting harder and harder to modify state-of-the-art solvers for controlled experiments or to try out new ideas. With CaDiCaL we tried to change this, thus the main objective was to produce a clean solver, with well-documented source code, which is easy to read, understand, modify, test, and debug, without sacrificing performance too much.

The first goals were achieved from the beginning and performance improved over the years. After its introduction in 2017 CaDiCaL continued to achieve high rankings in yearly SAT competitions, e.g., in 2019 it solved the largest number of instances in the main track, but scored less than the winner. It never won though except for the most recent SAT competition in 2023 where CaDiCaL was combined with a strong preprocessor employing bounded variable addition [55, 82]. The competition organizers paraphrased this as “CaDiCaL strikes back”.

Moreover, with the show-case of our new incremental approach [39] we invested in increasing the feature set supported by CaDiCaL culminating for now in supporting “user propagators”. This for instance allowed to replace the original but highly modified MiniSat based SAT engine in cvc5 by CaDiCaL, as described in a recent well-received SAT’23 paper [40].

The users of CaDiCaL fall into three categories. A first group applies the solver out of the box on benchmarks where CaDiCaL turns out to have superior performance. As an example consider solving mathematical problems with the help of SAT solving such as [78, 91, 108, 114]. Second, there is an increasing user base, including [6, 11, 23, 39, 40, 65, 92, 95, 101], which relies on the rich application programmable interface (API) provided by CaDiCaL, particularly its incremental features. Third, there are research prototypes modifying or extending CaDiCaL to achieve new features, including [7, 17, 43, 55, 64, 71]. Some of these modifications have been integrated [44, 100] but others remain future work [55].

Finally, CaDiCaL is used as a blue-print for understanding, porting, and integrating state-of-the-art techniques into other solvers. In this regard we are in contact with companies in cloud services, hardware design, and electronic design automation. It was also consulted in developing IsASAT [45], the only competitive fully verified SAT solver. Furthermore CaDiCaL was adopted as template solver for the “hack track” of the yearly SAT competition since 2021 as an “easy to hack” state-of-the-art SAT solver.

Related SAT solvers in the SAT competition often lack documentation, are hard to extend and modify, and, most importantly, do not provide such a rich and clean library interface as CaDiCaL. For instance our SAT solver Kissat [18] falls into this category. It has been dominating the SAT competition 2020–2022 (in 2022 all top-ten solvers were descendants of Kissat), is more compact in
memory usage and often faster on individual instances, but is lacking support for even the most basic incremental features such as assumptions.

The majority of the solvers in the SAT competition are restricted in their feature set as they are tuned for stand-alone usage, i.e., running the solver on a single formula stored in a file in DIMACS format [76], even though there is occasionally an incremental track in the SAT competition (last one that really took place was in 2020 as the one announced in 2021 was later cancelled).

Prominent SAT solvers with a richer feature set and particularly supporting incremental solving, beside the rather out-dated MiniSat [37], are newer versions of CryptoMiniSat [106], and GLucose [4]. The former is actively developed and in terms of implemented techniques has quite some overlap with CaDiCaL. In addition it offers special support for XOR reasoning, solution sampling and model counting [105]. The GLucose solver has been improved for incremental solving [3] but is not comparable in terms of implemented techniques nor features.

Unique and non-common features of CaDiCaL include: literal flipping [23], single clause assumption [46], incremental solving without freezing [39], extensive logging support, record & play of API calls, model-based testing, internal proof and solution (model) checking, termination and clause learner interfaces, various preprocessing techniques, an online proof tracing interface, formula extraction (after simplification), support of many external proof formats (DRAT, LRAT, FRAT, VeriPB) [100], and last but not least the user propagator [40].

This paper is structured by describing in the next section the architecture of CaDiCaL, which also acts as a summary of integrated techniques and provided features. The rest of the paper consists of highlighting recently added features of the solver or features not presented before, followed by experiments showing that CaDiCaL has state-of-the-art performance, before concluding.

2 Architecture

CaDiCaL is a modern SAT solver with many features written in C++. It can be used as stand-alone application through the command-line interface (CLI) or as library through its application programming interface (API) in C++ (or in limited form in C). Figure 1 depicts a structural overview. The central component, called Internal, implements CDCL search [83,103] and formula simplification techniques [24,69]. On top of it, the External facade hides the internals while maintaining the proofs and solutions (aka models) of solved problems.

The heart of the solver is the function cdcl_loop_with_inprocessing in Internal which interleaves the CDCL loop with formula simplification steps (i.e., with inprocessing [69]). During Search, CaDiCaL supports several techniques, like chronological backtracking [84,88], rephasing [32], and shrinking [44], which are only some of the important features. See Fig. 1 for more references.

The CDCL loop [83] is scheduled to be preempted in regular intervals to let the solver apply various formula simplification [24] and inprocessing techniques [69]. Each technique is implemented separately (e.g., in file subsume.cpp)
and has (i) a corresponding function which determines if the solver should pre-emt CDCL search and apply the technique (e.g., `subsuming()`) and (ii) a function that actually applies the technique (e.g., `subsume()`).

As Fig. 1 shows, CaDiCaL implements a variety of preprocessing/inprocessing techniques, including bounded variable elimination (BVE) [36], arguably the most effective one. As further examples, CaDiCaL also supports vivification [79, 98] and instantiation [1]. Combining them [22] won the CaDiCaL “hack track” 2023.

The External component communicates with Internal by mapping active variables into a consecutive sequence of integers (compacting) and extends internal solutions back to complete solution of the input problem with the help of the reconstruction stack [67]. In incremental use cases External also keeps the reconstruction stack clean [39] by “undoing” previous inprocessing steps. Beyond that, External connects internal and external proof generation (see Sect. 4).

We distinguish two types of API usage in CaDiCaL: static and dynamic. The static API provides access to standard solver functionalities between SAT solving calls (like IPASIR [8], parsing DIMACS, or iCNF files). With ILB as proposed by INTELSAT [86], we try to keep the trail unchanged between incremental calls.

The dynamic API interacts and controls the solver during Search. The solver provides dynamic access to clauses learned during conflict analysis to connected Learner instances. The Terminator class interface allows users to asynchronously terminate the solving procedure. Through the Iterator inter-
face of CADiCAL, the user can iterate over the irredundant (simplified) clauses of the problem or can iterate through clauses on the reconstruction stack, supporting simplified formula extraction and external model reconstruction.

3 External Propagator

Applications of CADiCAL, for example within the SMT solver cvc5 [11] (and maybe in the future within other lazy SMT solvers, such as Z3 [85] or Yices [35]), or to support Satisfiability Modulo Symmetries (SMS) [40, 116], require more control over the solver than provided by the standard incremental IPASIR interface [8]. To this purpose CADiCAL supports a more fine-grained and tighter integration into larger systems by allowing an external user propagator [26, 48, 49] to be connected to it through the IPASIR-Up interface [40].

This abstract interface is defined in the ExternalPropagator class which provides corresponding notification and callback functions. Inheriting from this class allows users to implement dedicated external propagators which for instance import and export learned clauses or suggest decisions to the SAT solver. The full description of functionalities supported by the IPASIR-Up interface is available in [40]. Here we focus on CADiCAL-specific implementation details.

First, CADiCAL ensures that only external variables appear in the IPASIR-Up interactions, thereby allowing users to ignore the internal (compacted) details. Furthermore, CADiCAL employs preprocessing and inprocessing even when an external propagator is connected. To avoid the need to restore clauses during the CDCL loop and to ensure that solution reconstruction [67] does not change assignments of observed variables (i.e., relevant to the external propagator), every observed variable is automatically frozen. As a side effect, the external propagator can only set clean [39] variables as new observed variables during search. As fresh variables are always clean, this is acceptable and mostly sufficient in practice.

Finally, CADiCAL, by default, considers every external clause as irredundant, exactly as the original input clauses of the problem. Thus, during clause database reduction they are not candidates for removal and so can be deleted only when implied by the rest of the formula. In future work we plan to allow users to specify the redundancy of the external clauses and to support incremental inprocessing [39] even for variables observed by the external propagator.

4 Proofs

Unsatisfiability proof certificates are an integral part of SAT solving [59, 60]. Even though clausal proofs were introduced in 2003 [37, 52], checking large proofs only became viable with deletion information [58]. The most prominent format today is DRAT [111] which was mandatory in the SAT competition from 2016 [10] to 2022. In 2023 both DRAT [111] and VeriPB [28] were allowed in the competition [9].
The proof formats GRAT [75] and LRAT [34] were proposed to allow even faster proof checking, i.e., by trading time for space, but also to facilitate formally verified proof checkers (e.g., CAKE_LPR [110]). They require hints for clause additions in form of antecedent clause identifiers (ids). External tools like Drat-Trim [111] can add such hints in a post-processing step to DRAT proofs.

The proof formats DRAT [111], FRAT [7], LRAT [34], and VeriPB [28] are supported by CADiCAL. It is the first solver to support LRAT natively. Without the need for post-processing this reduces proof checking time [100] substantially.

Recent diversification of proof formats in the SAT competition [9] motivated us to add VeriPB. It is a general proof format for various applications [28,50,51]. The tool-chain for checking SAT solver proofs with the verified VeriPB backend [28] is under development and not fast enough yet. Actually, BreakID-Kissat [9], one of the top performers in the SAT competition 2023, lost due to multiple timeouts during proof checking. Similarly to FRAT, CADiCAL can provide antecedents in VeriPB proofs. We expect this to speed up VeriPB proof checking considerably.

5 Tracer Interface

The dynamic API allows to extract proof information from CADiCAL online without files by connecting user-defined tracers as instances of the virtual C++ class Tracer. It provides notifications and callbacks for proof-related events, such as addition and deletion of clauses. Proof writers for all formats (Sect. 4) as well as both internal proof checkers (Sect. 8) go through the Tracer class. Furthermore, there is ongoing work to produce VeriPB proofs for the MaxSAT solver Pacose [97] using the Tracer interface in CADiCAL.

We support a large set of event types covering a multitude of use cases. Information provided includes antecedent ids and literals of clauses, separation between original, derived, and restored [39] clauses, and information of clause redundancy, as well as weakening [69] and strengthening [28,69]. For example, Fig. 2 shows the callback function for the proof event of adding a derived clause, where “derived” means entailed by the formula (i.e., not original input clause). Additional notifications include reserving ids for original clauses, as used for generating file based proof formats, such as VeriPB and LRAT.

For each solve call, a concluding event gives precise information about the result: a model concludes satisfiable instances, whereas for unsatisfiable instances we provide information about the final conflict clause. We have recently started to explore incremental proof tracing as well [41,42].

```c
void add_derived_clause (uint64_t new_id, bool redundant,
const vector<int> & literals_of_clause,
const vector<uint64_t> & antecedent_ids);
```

Fig. 2. Tracer virtual callback function to add a derived clause to the proof.
6 Constraints and Flipping

SAT solvers are used in a wide range of applications in many different ways. For incremental solving, MiniSAT has been the predominant choice. However, in recent years, CaDiCAL has begun to replace MiniSAT in numerous applications, most prominently cvc5. This can be attributed to its overall better performance and various application-specific features unique to CaDiCAL.

A prime example is the constraint feature [46], which allows users to define a temporary clause with the same lifespan as assumptions. It was initially developed to support the SAT based model checking algorithm IC3 [29], which requires often millions of incremental SAT calls during a single run, where each query needs to assume a single clause valid only for that call.

Constraints do not introduce new functionality per se, as temporary clauses can be simulated by activation literals. But they do allow the solver to employ a more efficient implementation, as they particularly avoid to introduce those assumption variables. Beyond IC3, constraints have also proven useful in our backbone extractor CadiBack [23]. The purpose of using constraint in backbone extraction is to find maximally diverging models in order to eliminate backbone candidates fast. CadiBack uses constraints to ensure that each new model includes at least one literal not observed in previous models. If this is not possible, all unseen literals are immediately determined to be in the backbone.

Once a model is found, we use another feature called literal flipping [19] to eliminate further backbone candidates [23]. A literal is flippable if toggling its value also results in a model. This concept was employed to speed-up backbone MiniBones [66] before and also MUS extraction [13]. In these earlier works it was implemented by iterating over all clauses outside the SAT solver, searching for literals that can be flipped in the model provided by the solver. Using clause watching our implementation inside of CaDiCAL is much more efficient.

7 Interpolation

Software-based test generation targeting RISC-V in the Scale4Edge project [38] relied on interpolation-based model checking and MiniCraig to generate interpolants. It uses MiniSAT as SAT solver and in this application constitutes a performance bottleneck. Therefore we developed a new more scalable solver CaDiCraig based on CaDiCAL and its proof tracer API (Sect. 5).

The implementation of CaDiCraig is external to CaDiCAL. It uses the same interpolant construction as in MiniCraig but is now seperated from MiniSAT. We are not aware of any other modern open-source SAT solver which allows to build interpolants through a generic API without being forced to write the whole proof to a file, trimming and post-processing it on disk, such as in [53].

The CaDiCraig tracer constructs partial interpolants as usual, e.g., see [73]. Through the proof tracer API the tracer is notified by CaDiCAL about each new clause and its antecedents needed to derive it by resolution. It then builds a partial interpolant for that clause using previously computed partial antecedent
interpolants. When the solver concludes deriving an empty clause and thus showing unsatisfiability (Sect. 5) the final interpolant is built from the antecedents of the empty clause. It can then be retrieved by via the CaDiCraig API.

8 Testing and Debugging

Such a sophisticated and complex software as CaDiCaL necessitates rigorous testing to ensure correctness of interactions between its multitude of features. In this section we discuss our arsenal of essential testing and debugging techniques.

First, we primarily rely on logging for debugging purposes. For instance, when enabled, CaDiCaL will print every single step from its creation to its deletion. From an implementation perspective, logging features are not compiled in by default to avoid performance overhead in release builds. Furthermore, if enabled at run-time, CaDiCaL prints verbose information about the inprocessing schedule, useful for debugging performance regressions (e.g., inprocessor scheduling).

Further useful debugging tools are the built-in checkers. The LRAT and DRAT checkers are optional and ensure that every learned clause is properly derived. The new LRAT checker [100] was crucial for achieving LRAT support.

Last but not least we want to mention the API fuzzer Mobical, which generates random API calls and minimizes failing runs. Internally, Mobical implements a state machine issuing API calls. It also performs option fuzzing by varying available options. This approach is extremely useful to produce short failing API call traces focusing on the actual defect, e.g., like picking a low garbage collection limit to trigger a defect in the garbage collector. Combining checkers with Mobical greatly increases its strength. During development it is advisable to build Mobical and CaDiCaL with assertions and checkers enabled.

Mobical is similar in spirit to the related model-based tester of Lingeling [2] for SAT and BtorMBT [94] and Murxla [93] targeting SMT. Note that other SMT fuzzers [27,30,96,102] focus on non-incremental usage or only support incremental "push & pop" [80]. For non-incremental SAT solving, there is also cnffuzz fuzzer and the cnfdd delta-debugger [2,31].

Accordingly, we have implemented a MockPropagator class in Mobical to test the EXTERNALPROPAGATOR API. It fuzzes the Ipasir-Up implementation in combination with all options and features of the solver. It revealed several corner-cases which we believe would have been very hard to trigger otherwise.

Mobical targets only incremental SAT problems and could not help when incorrect interpolants showed up in earlier experiments with MiniCraig and CaDiCraig. Therefore, we have built an external interpolation fuzzer in Python. It checks interpolants and an accompanying delta-debugger minimizes problems by deleting command line options, clauses, and variables.

9 Experiments

The performance of CaDiCaL 2.0 was evaluated in three experiments. We first follow the non-incremental setup of the main track of the SAT competition,
where solvers are run on benchmark files in DIMACS format. The second experi-
ment focuses on incremental usage, i.e., following the incremental track of the
competition. Finally we show the effectiveness of CaDiCaL in the context of
interpolation via its Tracer API. All experiments were conducted on our cluster
with Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (turbo-mode disabled).

Non-Incremental. The winner SBVA-CaDiCaL [54] of the main track of the SAT
Competition 2023 combined a novel technique for bounded variable addition [82]
with CaDiCaL 1.5.3. In their implementation preprocessing was limited to 200 s
which yields different preprocessed formulas over multiple runs. Therefore, we ran
the preprocessor of SBVA-CaDiCaL separately for 200 s, and then gave the same
formulas to CaDiCaL 1.5.3 and our new version CaDiCaL 2.0. Running them
for 5000 s as in the competition (ignoring preprocessing time in essence) gave
very similar results. We provide more details in the artifact. This confirms that
CaDiCaL (also in version 2.0) is state-of-the-art in non-incremental solving.

Incremental. How to assess the incremental performance of a SAT solver is less
established. To present an unbiased evaluation, we follow the principles set out
by the last incremental track of the SAT competition in 2020 [47]: The solvers
are evaluated in six different applications, each featuring 50 benchmarks, with
a 2000 s timeout and 24 GB memory limit. Four applications are carried over
directly from the 2020 competition: the CEGAR-based QBF solver Ijtihad,
the simple backbone extractor Bones, the longest simple-path search LSP, and
the MaxSAT solver Max. However, we exclude two applications: the essential
variable extractor and the classical planner Pasar. Both use features that are
not present in all solvers. The former queries ipasir_learned, which is missing
from CaDiCaL 1.0, and the latter relies on limiting the number of conflicts.
Instead, we include the bounded model checker for bit-level hardware designs
CAMiCaL [39] and the sophisticated backbone extractor CaDiBack [23].

The benchmarks from the incremental track from the 2020 SAT Competition
remain unchanged. For CAMiCaL, we randomly select 50 Boolean circuits used
in HWMCC’17 [25]. Although CaDiBack solves the same problem as Bones,
we opt for a distinct set of benchmarks. In 2020 the “smallest and easiest satis-
ifiable” [47] CNF formulas were selected and even though backbone extraction
is harder than mere solving, they were rather easy. Conversely, we compile a
non-trivial set of benchmarks by randomly selecting satisfiable formulas from
past competitions (2004–2022) [23] that take Kissat 3.0.0 [19] more than 20 s
to solve. We use Kissat as it is not incremental and hence does not compete.

The artifact has a comparison of CryptoMiniSat and CaDiCaL on 1798
formulas [23] and indicates that our selection does not impact the outcome. As
detailed in Sect. 6, CaDiBack utilizes constraints, which are only available in
recent versions of CaDiCaL and are simulated with activation literals otherwise.

Our evaluation includes all solvers that competed in 2020: Riss 7.1.2 [81],
CryptoMiniSat 5 (CMS) [104,107], and abcdSat i20 [33]. The CaDiCaL
version from that year is referred to as CaDiCaL 2020. The other two versions
are 1.0 from 2019 and our latest release 2.0. We also include MiniSat 2.2 and
the latest version of Glucose 4.2.1. Table 1 presents for each SAT solver and application: the PAR2 score, which is the average runtime in seconds with a penalty of 4000 for unsolved instances; and the number of solved instances.

**Table 1.** Performance comparison of six incremental solvers, with three versions of CaDiCAL (2000 s timeout). For each solver, we report PAR2 score over 50 benchmarks per application and number of solved instances (“PAR2 | #solved”). The four applications to the right have been used in the incremental track of the 2020 SAT competition. The best results per application are marked in **bold**. The last row presents the hypothetical Virtual Best Solver which always picks the best performing backend for each instance.

<table>
<thead>
<tr>
<th></th>
<th>CaDiBack</th>
<th>CaMiCaL</th>
<th>Bones</th>
<th>LSP</th>
<th>Max</th>
<th>Ijtihad</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaDiCAL 2.0</td>
<td>3297</td>
<td>2606</td>
<td>494</td>
<td>1898</td>
<td>1976</td>
<td>2980</td>
<td>2209</td>
</tr>
<tr>
<td>2020</td>
<td>3409</td>
<td>2677</td>
<td>622</td>
<td>1955</td>
<td>2015</td>
<td>2986</td>
<td>2277</td>
</tr>
<tr>
<td>1.0</td>
<td>3495</td>
<td>2627</td>
<td>595</td>
<td>2011</td>
<td>2028</td>
<td>2989</td>
<td>2291</td>
</tr>
<tr>
<td>CMS</td>
<td>3491</td>
<td>2701</td>
<td>397</td>
<td>1773</td>
<td>2021</td>
<td>3057</td>
<td>2240</td>
</tr>
<tr>
<td>MINISAT</td>
<td>3678</td>
<td>2807</td>
<td>687</td>
<td>1993</td>
<td>2094</td>
<td>3123</td>
<td>2397</td>
</tr>
<tr>
<td>RISS</td>
<td>3665</td>
<td>2836</td>
<td>892</td>
<td>1835</td>
<td>2017</td>
<td>3140</td>
<td>2398</td>
</tr>
<tr>
<td>ABCDSAT</td>
<td>3582</td>
<td>2966</td>
<td>535</td>
<td>2493</td>
<td>2037</td>
<td>3207</td>
<td>2470</td>
</tr>
<tr>
<td>GLUCOSE</td>
<td>3778</td>
<td>2981</td>
<td>948</td>
<td>2078</td>
<td>2117</td>
<td>3206</td>
<td>2518</td>
</tr>
<tr>
<td>VBS</td>
<td>3127</td>
<td>2546</td>
<td>257</td>
<td>1765</td>
<td>1856</td>
<td>2896</td>
<td>2075</td>
</tr>
</tbody>
</table>

Our results show that CaDiCAL 2.0 reaches state-of-the-art performance, demonstrating a distinct improvement over previous versions. Also, differing from the findings in [72], we see a significant advantage of the newer CaDiCAL and CryptoMiniSat, over the older MiniSat, further substantiated below.

**Interpolants.** To validate CaDiCraig using CaDiCAL, we converted all 400 benchmarks of the SAT Competition 2023 into interpolation problems split into A and B parts chosen with the goal to assign related clauses to the same part in order to keep the number of global variables limited. The index of the smallest variable of each clause determines the probability of the clause being assigned to A. On our crafted benchmarks (5000 s timeout, 7 GB), CaDiCraig significantly outperforms MiniCraig, solving 117 benchmarks, compared to only 75.

**10 Conclusion**

In this very first conference paper on CaDiCAL we reviewed its most important components and features as well as its testing and debugging infrastructure. We highlighted its use as SAT engine in SMT solving via the user propagator interface and how the tracer API can be used to compute interpolants. Our experiments show that CaDiCAL remains efficient despite this flexibility.

Producing incremental proofs is ongoing work [41,42]. Further future work consists of producing incremental proofs for all features supported by CaDiCAL,
avoiding to freeze observed variables by the user propagator, and porting into
the main branch features provided by other users.

Acknowledgements. This work was supported in part by the Austrian Science
Fund (FWF) under project T-1306, W1255-N23, and S11408-N23, the state of Baden-
Württemberg through bwHPC, the German Research Foundation (DFG) through grant
INST 35/1597-1 FUGG, the German Federal Ministry of Education and Research
(BMBF) within the project Scale4Edge under contract 16ME0132, and by a gift from
Intel Corporation.

References

combinational design automation problems. In: Proceedings of the 39th Design
Automation Conference, DAC 2002, New Orleans, LA, USA, 10–14 June 2002,
Heidelberg (2013). https://doi.org/10.1007/978-3-642-38916-0_3
solving with assumptions: application to MUS extraction. In: Järvisalo, M., Van
(2013). https://doi.org/10.1007/978-3-642-39071-5_23
Conference on Artificial Intelligence, Pasadena, California, USA, 11–17 July 2009,
ijcai.org/Proceedings/09/Papers/074.pdf
S0218213018400018
6. Bacchus, F.: MaxHS in the 2022 MaxSat evaluation. In: Bacchus, F., Berg, J.,
and Benchmark Descriptions. Department of Computer Science Series of Publica-
7. Baek, S., Carneiro, M., Heule, M.J.H.: A flexible proof format for SAT solver-
Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_4
(2016)
SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions.
Department of Computer Science Series of Publications B, Department of Com-
puter Science, University of Helsinki, Finland (2023)
and Benchmark Descriptions. Department of Computer Science Series of Publica-
Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24


79. Luo, M., Li, C., Xiao, F., Manyà, F., Lü, Z.: An effective learnt clause mini-


Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.