SAT solvers: Verify, Improve, and Use Them in Interactive Theorem Provers

universität freiburg

Written Summary of the Scientific Publications Submitted for the Habilitation of

Mathias Fleury

Draft for Part I and Part II

Universität Freiburg, Freiburg Im Breisgau, Germany
May 6, 2024

Abstract

Ensuring the correctness of systems is very important to ensure the safety and functionality of those systems. Many verification approaches rely on SAT solvers either by translating the problem to check and asking a SAT solver or by translating only partially to a SAT solver and slowly refining the translation, like SMT (Satisfiability Modeling Theory) solvers.

The first contribution from this thesis is about understanding and improving SAT solvers. We benchmark old SAT solvers from the early 90s and observe the performance improvement. Solvers have become better and better over the years on both old and new benchmarks. We also improve SAT solvers not only by changing the way the search direction is determined but also by extracting more information from the input, like in the state-of-the-art SAT solver Kissat. Here is the text with typos corrected:

However, verification by translating and trusting another system is unsatisfactory. Instead, we use certificates that we check independently. We analyze those from our parallel SAT solver Gimsatul. Interestingly, we observe that the proof size is independent of the number of threads. However, checking is very costly, as the checker still needs to search for the derivation of the new information (unlike the SAT solver that both has to derive and find the new information). Therefore, we extend our state-of-the-art single-threaded SAT solver CaDiCaL to produce an enriched proof that is checkable without search. Beyond, we also work on a proof format beyond SAT to verify hardware multipliers. The verified proof checker Pastèque verifies the proofs produced by Amulet. We verified it in the Isabelle proof assistant.

Proof assistants verify each step of a human-produced proof. They check if programs match their specifications. we have fully verified the SAT solver IsaSAT. And now, we have added simplifications that occur during execution and improved it by changing the target programming language. While actually increasing trust by removing a trusted (unverified) level in the translation.

Finally, we are working on the collaboration between SMT solvers and the theorem prover Isabelle. The proof assistant is skeptical but it can check the proofs produced by the SMT solvers (instead of asking the user to produce one). We implement reconstruction for proofs produced by veriT, and are now working on cvc5, which gives more details and produces proofs for more expressive logics, like bit-vectors.

Contents

List of Figures

Part I

Written Summary of the Publications Submitted for Habilitation

1 Introduction

The accuracy of computers is critical as human reliance on automated systems grows. A notable example of things going awry is the Intel Pentium bug, or the FDIV bug, a design flaw in the floating-point unit (FPU) of Intel's 1994 Pentium processor that resulted in inaccurate results for certain floating-point division operations.

One way to prevent the FDIV bug is exhaustive testing. For 32-bit numbers, it would nowadays be possible to do so, because $2^{32} \times 2^{32}$ is approximately 9 billion cases which at current CPU speed of around 5 GHz is easily doable. However, this was not an option in 1994 with frequencies of 66 MHz and the testing approach is not feasible for 64-bit numbers anyway. Therefore, we need formal verification to check that our systems are correct.

On way to achieve reliability is through a mechanical proof of correctness, which involves validating every execution path, including to ensure no memory corruption, even during an entire year's run (untestable through standard testing). However, designing such systems is complex, reducing flexibility for various options, and typically results in slower performance compared to unverified systems.

Overall, we distinguish six levels of trust, going from the minimum (testing, which can still miss bugs) to perfectly correct systems (not achievable today):

1. Carefully designing the system with code review, testing, and documentation.

2. Translating the current problem and asking formal tools with multiple backend engines (like SAT solvers) for a proof or counterexample and waiting for one of them to claim that the system to verify is correct.

3. Using a formal tools producing a proof checked independently by another non-verified system, where the input is the system to verify.

4. Using a formal tools producing a proof that is checked by a fully verified system.

5. Completely verifying the formal tool, ensuring that every possible input problem yields a correct result.

6. Running checkers and verified solvers on fully verified hardware (running them several times on different machines to take into account environmental factors like random bit flipping due to solar radiation).

Since the FDIV bug, as far as publicly known, Intel has withdrawn from Level 1 and now mainly employs Level 2 (although some of the tools used internally are able to produce proofs, like SAT solvers). Level 6 remains unattainable today.

The work presented here begins with a formal engine at Level 2: If the formal engine is not good enough, it makes no sense to care about proofs. One of my focus areas is on understanding and improving SAT solvers. They handle a simple input language (propositional logic) without integers or quantifiers, but this problem is already NP-complete. These solvers form the foundation for more intricate systems such as bounded-model checking and SMT solvers.

Even if SAT solvers are tools used in the Level 2, the development changed over time along the six steps of verification. In 2010, Brummayer, Lonsing, and Biere [19] found that nearly all SAT solvers were buggy by fuzzing them (i.e., generating random inputs). Therefore, to increase trust in SAT solvers, the community they went to Level 3 with the production of certificates that can be checked. It took a while before the right proof format (easy to generate, expressive enough, and not too hard to check) and the DRAT format became mandatory in the SAT Competition. Then, the proof checkers were verified (and some bugs were found in the non-verified one, although I am aware only of crafted counter-examples). Nowadays, there is again a push to generalize the proof format in order to support more general steps in the proof.

My first contribution involves understanding and enhancing SAT solvers, including tracking their progress. There is a yearly contribution, but progress is not tracked through annual competitions. In collaboration with Biere, Froleyks, and Heule [H2][1], we analyze the performance of SAT solvers on older and newer benchmarks. Our findings reveal that more recent solvers solve significantly more problems – both on old and on new benchmarks. Therefore, the progress is not just about better performing on the newest benchmarks at the cost of the old ones. Besides this, I also worked on improving the performance by extracting lost information during translation to the input proof format and reducing memory usage (Chapter 4).

My second contribution involves the certification produced by an SAT solver. To speed up verification, verified checkers rely on hints that must

[1] To make the distinction easier, the publications included in the Habilitation are prefixed with the letter 'H'

be found. With Pollitt and Biere, we extended the SAT solver CaDiCaL to directly produce the hints without performance loss. I also worked on the parallel solver, Gimsatul. With Biere, we discovered that proof size was largely independent of the number of threads. In addition to my work on certificates, I collaborated with Kaufmann on developing a verified checker for verifying multipliers based on handling polynomials (Chapter 3).

The trade-off is that each problem's result must be checked (with verified checkers at Level 4) instead – and DRAT checking sometimes takes more than solving. My third contribution builds upon my PhD thesis with various improvements of my fully verified SAT solver IsaSAT (Level 5). This brings SAT solving to Level 5. SAT solvers typically consist of two primary components, the solving (CDCL) engine and a simplification engine. My SAT solver, IsaSAT, now incorporates both, though it supports fewer techniques than other unverified state-of-the-art solvers. Additionally, I modify the target language of synthesis of IsaSAT to the intermediate language used by LLVM. This significantly improves the performance.

To verify such a system, I employed the interactive theorem prover Isabelle. It relies on user inputs to guide proof discovery by providing complex invariants and properties, intermediate proof steps, while the interactive theorem prover ensures the proof is correct (by trusting only a small kernel) and all cases have been covered. In the case of IsaSAT, I proved that the solver terminates, potentially with an unknown answer if a lot of memory is used (Chapter 4).

The main challenge of full verification lies in its time-consuming nature and lower performance compared to the best solvers. My final contribution involves integrating interactive verification and proof generation by enabling the use of SMT solver proofs in theorem provers. In collaboration with Schurr and Desharnais, we incorporate the SMT solver veriT into Isabelle. Additionally, I work with various people to establish a standard for the proof language, Alethe, which is now used by the SMT solver cvc5 for its proof format (with additional proof details). cvc5 seeks to further enhance their system by incorporating user-defined rules, called RaRE rules, during compilation. This feature simplifies the process of extending the solver. To facilitate this integration, I supervise Hanna Lachnitt's work on automatically translating RaRE rules to Isabelle (but most of them still require a proof!). While the complete integration of cvc5 into Isabelle is ongoing, the set of RaRE rules has not yet been finalized (as discussed in Chapter 5).

In subsequent chapters, I delve deeper into each topic, discussing main ideas and results. For more information, consult the corresponding publications.

2 Understanding and Improving SAT Solvers

SAT solvers check for satisfiability of a problem in propositional logic with some restriction on the input. The input format is based on a concatenation of clauses. Each clause is a disjunction of literals, and each literal is either a positive or a negative variable. The SAT problem is about either finding a model, namely a mapping such that in each clause the model assigns at least one literal to true, or proving that no such model exists. For example, if the clauses are $a \vee \neg b \vee c$ and $\neg a$, then a (partial) model is $\neg a, c$. A total assignment would be $\neg a, b, c$, assigning a value to all variables.

SAT is the traditional NP-hard problem that interests many theoreticians: after guessing the problem with an oracle, checking that a model is correct is polynomial in the input (as it is sufficient to iterate over all clauses). All other NP-complete problems can be translated in polynomial time to SAT. In practice, however, SAT solvers can solve many problems efficiently.

The performance of SAT solvers has significantly improved over time, enabling them to solve problems with millions of variables that were previously unattainable, even if some smaller ones remain out of reach for the best solvers. SAT solving started in the 60s with DP [27] and DPLL [26].

SAT solvers consist of three main components: search, reduce, and simplify rounds. During the search phase, a problem is typically deemed satisfiable or unsatisfiable. Solvers spend most of the time in search. The search algorithm, called CDCL [21], is the most important part of any implementation to solve the SAT problems.

The search process constructs a partial assignment called trail, propagating current information or guessing values. Propagation happens when there is a single extension of the partial assignment that is compatible with a clause. For example, if the clause $a \vee b$ and the partial assignment contains $\neg b$, then the only way to satisfy the clause is to set a to true. The clause $a \vee b$ is the *reason* of the propagation. If the partial assignment becomes incompatible with the formula (like in the previous if the assignment also contains $\neg a$, it is impossible to satisfy $a \vee b$), then *backjumping* adapts it. Earlier solvers from the 60s would only switch the last decision, while subsequent improvements

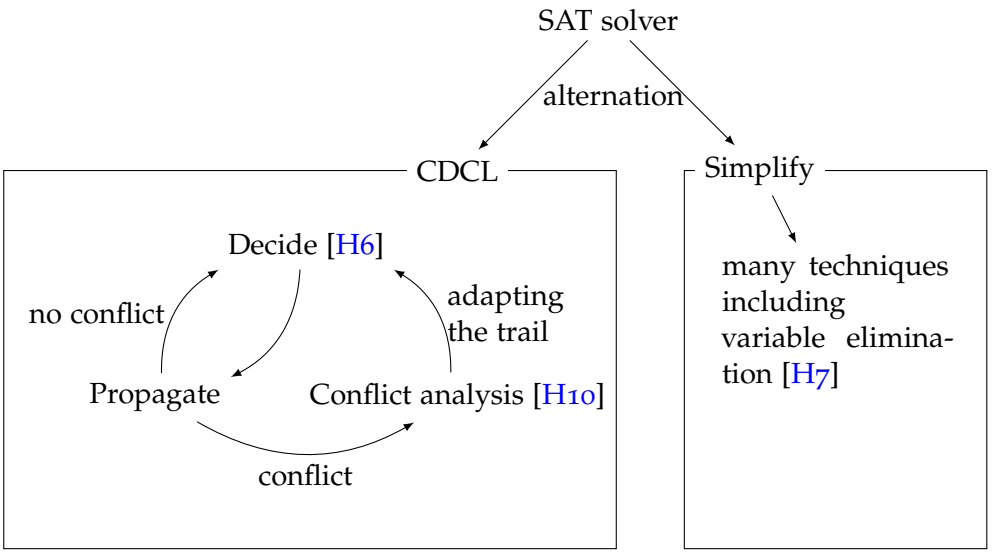


Figure 2.1: Abstract representation of the different parts of a SAT solvers

like CDCL [25] actually learn new clauses that explain how to fix the trail, preventing the solver from encountering the same dead-end again.

Search produces thousands of new clauses per second. The solver discards most of these to reduce memory pressure during reduction rounds. While the clauses might still be useful to reduce the search space, the solver deems some as not useful. There are many heuristics for this, the most efficient seems to be the LBD score [20], giving each clause a score of how useful the clause is to propagate information. Nowadays, it is combined with a second chance algorithm [15]. Learning clauses makes the solver exponentially more efficient in theory, but SAT remains NP-complete and therefore exponential in the worst case. Additionally, even if a problem has a polynomial solution, it does not mean that a solver will find the optimal solution.

The simplification phase attempts to remove duplicated information in the clauses. It can do this by removing clauses, although all learned clauses are redundant and can be removed without harming the correct answer of the solver. The phase can also learn clauses in ways that CDCL is not able to, such as discovering that literals are equivalent. Many recent papers have focused on developing new techniques for this phase, including vivification, which appeared in 2016 and is based on an older idea called distillation [23], and ERE [7]. Most solvers, such as MapleSAT or Glucose, only perform simplification at the beginning. However, with the rise of CaDiCaL and

KISSAT, nearly all solvers now interleave search and simplification, known as inprocessing. There are two approaches: removing redundancy in the clauses (using techniques like subsumption, vivification, etc.) and deeper problem transformations like variable elimination, which eliminates one variable by producing all resolvents, or variable addition, which does the opposite. The former is the most important technique today, but the latter (as a preprocessing technique) won the SAT Competition 2023.

One crucial factor for satisfiability problems is the guess direction during search. If the problem is satisfiable, then the problem has a model, and the best choice for guesses is a value of that model. However, finding a model is exactly solving the NP-hard problem and hence is hard. Since the SAT solver MINISAT [24], the default direction is to set variables to the last assigned value and false if the variable was never set. While this choice improves performance in the SAT Competition (closed-world assumption), it makes finding solutions with all literals set to true nearly impossible. The largest problems in the SAT Competition are 1.5 GB files that are easy to solve with the current heuristics of defaulting to the false value.

Performance is best measured in the SAT Competition and SAT Races (either/or depending on the exact year), an annual event that identifies the fastest SAT solvers on a new set of benchmarks. Each SAT-solver author must submit some new benchmarks. Over the years, the results showed improvements, and subsequent solvers included techniques and ideas from the winners of the past years. For example, bounded variable elimination: it was initially included in a separate preprocessor SATELLITE, then included into MINISAT, and this technique is now included into (nearly) all solvers (to the best of my knowledge, only my verified SAT solver from Chapter 4 does not include it in the last SAT Competition).

Own Contributions

Evolution of SAT Solvers [H2]. To review the history of SAT solvers, we gathered all winners from the SAT Competition since 2002, as well as some historically significant solvers. We ran these solvers on benchmark sets from 2002, 2011, 2021, and 2022. This large benchmark set is to ensure that performance are not only due to adapting strategies on the newest benchmarks without any improvement.

This project consisted of three main steps. First, we obtained the source code for each solver. Recent ones were easily accessible from the SAT Competition webpage. Older solvers required contacting their authors (who usually

only had it in some computers or backups). Second, we modified the code to ensure compatibility with modern compiler versions, addressing issues such as 32-bit integer operations still functioning on 64-bit systems. We solved one nasty bug in CryptoMiniSAT by using stable sorting instead of standard methods (the bug was probably already there, but it was not visible, because it depended on the implementation in the C++ standard library). Lastly, we verified consistent results across all runs.

A limitation of this approach is the need to restore code in the future when compilers become more strict regarding C or C++ code. We even noticed some differences between gcc-9 and gcc-11 versions. Unfortunately, one solver proved unfixable. We also do not attempt to fix all issues present in the source code (by using fuzzing or producing proofs) and it is unlikely that the old solvers are error-free. Some of the issues we encountered led to solvers ignoring part of the input files, so they were solving a different problem.

There is some bias in the benchmarks from the competition, because solvers have been trained on the SAT Competition. However, it is unlikely that recent solvers have been trained on old problems (2011 and before) and it is impossible that old solvers have been trained on new problems. The results over all four sets of benchmarks indicate significant improvements in SAT solvers over time, with consistent progress across the years. This demonstrates that advancements are not solely due to training on recent benchmarks but rather a combination of various factors, including improved heuristics and algorithms to have better scaling for larger problems.

There is another experiment by Fichte et al [11]. They tried new solvers on old hardware and new old solvers on new hardware. The first version of their work did not check for consistency of the results (and we found problems deemed both SAT and UNSAT depending on the solver). They have updated the work and they sorted out solvers that had *too many* incorrect results, but still included known-wrong solvers. Their experiments are not available (and they have not answered our question in private communication); so we cannot check ourselves, but comparing only correct results (and attempting to understand why some are wrong) is the minimum we expect from such an experiment. And the minimum you would expect when using SAT solvers for the Level 2 of verification.

Intriguingly, similar experiments in different contexts (MaxSAT and SMT solvers) reveal that SAT solver progress does not necessarily translate to improvements in related tools like MaxSAT and SMT solvers. In the latter case, one possible explanation is that most of the search time is not spent in SAT solvers, but for the former, it is less clear.

*Note: I added the missing citation reference to the experiment by Fichte et al.

Reducing Memory-Usage by Learnt Clauses [H10]. Reducing the size of learned clauses is crucial as the solver uses less memory and makes more frequent use of shorter clauses, which are more beneficial than longer ones. One method for achieving this is learned clause minimization [22]. It eliminates a literal from the conflict clause by resolving (rewriting) with reasons along the trail, if the resolution does not require adding any new literal to the conflict. In essence, the algorithm is doing rewriting along the reasons of the trail and it reduces a clause to a subset of it.

In 2020, a CaDiCaL hack by Feng and Bacchus [10] won the planning (application) track by a significant margin with a different approach: resolving along the trail only if it results in a smaller clause size, leading to much less memory usage, even if this requires adding new literals to the clause. Only the size of the end result is important. However, this solver did not perform well in the main track.

We implement a variant called *shrinking* that adds new literals only if it does not increase the LBD score of the clause (the standard approximation for usefulness in modern SAT solvers). We implement it efficiently to be compatible with the cache used by minimization. For each level, our implementation goes twice (once for shrinking and once for minimization if shrinking failed) over the clause, without erasing the cache used for minimization, reducing the complexity. As this technique relies on resolving clauses, it can generate proof using the standard proof format.

Additionally, we prove that the minimization algorithm is complete: it removes all redundant literals (redundant with respect to the clauses used by this technique, namely the reasons in the trail, not to all clauses of the input problem). We also test the more advanced minimization criteria to detect earlier when literals are not removable, although this did not affect performance much. However, we can formally explain why the criteria does not change the minimizable literals.

Overall, shrinking has minimal impact on the main track of the SAT Competition in the three SAT solvers we implemented it in. However, it had a significant impact on the planning track – but less than the original implementation that much more aggressively shortens clauses at the cost of worsening the score of the clauses (and performs worse on the main track). Its main advantage seems to be for problems containing many binary clauses: Shrinking reduces binary implication chains on one level to a single literal.

In theory, this could reduce the usefulness of the clause, but binary clauses are never deleted, so reducing a chain to its antecedent has no impact on usefulness (the subsequent propagation chain will always be possible, which is not the case if the propagation includes larger clauses that are regularly deleted).

Changing the Search Direction [H6]. Restarts are a crucial component of SAT solvers, enabling them to alter the current search direction by modifying ongoing guesses, thereby leading to different assignments. This feature is essential for discovering superior proofs from a proof-theoretical standpoint, and in practice, it helps avoid the heavy-tail phenomenon where the solver becomes trapped in an unsatisfiable portion of the search space that backtracking cannot easily escape without exploring the entire subspace.

In this paper, we propose a novel perspective on CDCL, viewing it as an optimization problem that involves maximizing the conflict-free segment of the trail. To accomplish this objective, we limit the number of restarts to allow for more extensive exploration of assignments and enforce the SAT solver to revisit the same section of the search space by consistently setting literals in the phase of the most optimal conflict-free assignment discovered thus far.

However, always forcing the solver to go into a single direction is detrimental to performance. Therefore, we alternate between the modes where we actively restrict the solver to go into the same direction and the usual mode where the solver uses the standard phase saving (search direction for each variable). In order to search into more directions, we use another method called rephasing. The solver periodically resets all saved phases and changes them. As this can be detrimental to CDCL, we implement rephasing in geometrically increasing intervals of conflicts to ensure determinism. Rephasing is also an effective solution for finding solutions to problems with all literals set to true or false since both directions are attempted.

During rephasing, we alternate between different goals: all true, all false, flipping, and assigning the phases to the best assignment found so far, intending to help the solver focus on the unsatisfiable part of the formula. If the formula comprises a satisfiable and an unsatisfiable segment, once a model for the satisfiable portion is identified, the solver will concentrate on the remaining formula (as a byproduct of heuristics that enhance variable importance). However, rephasing ultimately overwrites the assignment of the satisfiable part. Therefore, we implemented autarky detection to identify and remove solved portions of the formula in such cases.

Finally, both techniques facilitate the integration of CDCL solvers with local search (SLS), which involves flipping one literal in a full assignment if it is not a model. It can only deem a formula satisfiable. SLS solvers are poor at detecting propagation chains since they must flip the correct literals throughout the entire chain. We utilize the CDCL assignment as a starting point, given that propagation has already been carried out. Both CDCL solvers and SLS work together by using the assignment identified by the SLS solver as a search initiation point for CDCL.

Biere and I [9] and Cai and Zhang [6], independently proposed this idea of improving a partial assignment produced by CDCL with local search and using the improved assignment for CDCL again. Although the specific implementation varies (with one group remembering the best partial assignment and the other utilizing CDCL when enough literals are set and ignoring conflict to find a total assignment that serves as the starting point for the SLS solver), the concept is identical, and we all observed for the first time that SAT solvers perform better with this combination, enabling them to solve more problems than either of CDCL and SLS can individually.

Extracting More Information from the CNF [H7]. SAT solvers only take formulas in Conjunctive Normal Form (CNF) as input. However, this makes it hard to recover some information, such as definitions. The most important simplification technique is bounded-variable elimination (BVE). The idea is to eliminate one variable by resolving all clauses that include it and removing the clauses with that literal. Technically, this is a decision procedure, but the memory usage can explode before the problem is either reduced to the empty formula or the empty problem. Therefore, elimination is bounded to not increase the number of clauses (or only by a small margin).

Recognizing definitions does not help the CDCL part of the SAT solver, but it can improve the performance of bounded-variable elimination. Instead of resolving all clauses together, it is sufficient to resolve with the clauses that are the definitions. This reduces the number of new clauses and makes BVE more efficient.

For example, $a = (b \vee c)$ corresponds to the three clauses $\neg a \vee b \vee c$ and $a \vee \neg b$ and $a \vee \neg c$. With simplification, we might end up with the strengthened clause $\neg a \vee b$ and still have to recognize the definitions. Then instead of resolving all clauses together (defining and all other clauses), the solver only needs to resolve all other clauses with the defining clauses.

The usual way to extract definitions is to use syntactic criteria and recognize the definitions. However, this does not work for more complicated gates

and has issues with strengthened clauses. Therefore, we worked on a better definition extraction mechanism: We use a simple SAT solver, called Kitten, with a subset of the clauses (the environment).

For performance reasons, only a small environment is given to Kitten and the subsolver is only run for a small amount of time. If the problem is unsatisfiable, then the unsatisfiable core (i.e., the subset of all clauses that lead to concluding false) are exactly the clauses of the definition.

Our experiments show that deactivating Kitten reduces the number of eliminated variables (as expected), without much performance impact. Syntactic detection is still useful as it is much faster, and AND-gate detection is the most important for SAT Competition problems (in 2020, there are around 150 problems (out of 400) where at least 20% of the eliminated variables are defined by an AND).

Future Work

There is an increasing number of techniques that overlap and can simplify clauses in similar ways. A lot of the art of programming efficient SAT solvers involves combining all of them with a limited time budget, as nearly every technique is sometimes useless while requiring a huge amount of time.

We are currently working with Karem Sakallah and others to produce a modular solver in which we can test techniques independently of each other. We also want to explain why techniques work or don't. Interestingly, while attempting to produce scalable benchmarks, we found a performance regression in Kissat. We investigated it and found the difference to be one of the inprocessing techniques, vivification.

We are also working on various new techniques to extract information that is hidden in CNF files and on other techniques to improve solving time. One of the solvers where we have implemented most of the techniques is the SAT solver CaDiCaL, which I help maintain. CaDiCaL contains many techniques building on top of CDCL, such as a unit propagator where external reasons can be given, similar to how SAT solvers work.

It is, however, not sufficient to understand the different techniques; understanding the scheduling is critical. This is one of the major differences between Kissat and CaDiCaL, but it is not clear whether Kissat is faster thanks to this or despite of it.

3 Beyond Verification: Certification

The correctness of SAT solvers is critical when using them for verifying systems. Confirming a SAT answer is simple by iterating over clauses and checking if a literal is true in each clause. However, unsatisfiable (UNSAT) answers are challenging to check. One proposed solution is to use two independent SAT solvers. But compiler bugs can produce incorrect answers. Additionally, defining what independent solvers are is difficult because many SAT solvers have similar propagation loops. This issue occurred in the SAT Competition 2013, where a compiler bug in GCC led to incorrect answers. Footnote: ¡https://groups.google.com/g/minisat/c/YHEWCFqxyFg¿

The requirement of two solvers agreeing is helpful when proofs are unavailable. Fuzzing, which involves producing random input to test all execution paths in a solver, can also be used for comparison. If one solver produces a SAT result and the other produces UNSAT, it's easy to determine which solver is incorrect (by checking if the model is correct). Currently, there is no alternative for MaxSAT solvers, which, like SAT solvers in the past, have bugs and proof certificates are still very new. However, MaxSAT solvers are robust enough not to produce different results in the MaxSAT competition.

SAT's unsolved answers are verified via proof production. In the SAT Competition, only problems with confirmed proofs count as solved. The competition currently accepts two proof formats: DRAT and veriPB. To check DRAT proofs, DRAT-trim transforms the proofs into LRAT, which in turn the verified checkers cake_lpr or GRAT checkers validate. The veriPB checker verifies veriPB proofs.

Producing proofs in SAT solving has another advantage: we can gather information about the search process. One crucial piece of information is whether learnt clauses are useful or not. However, DRAT-trim relies on heuristics to determine the usefulness of a clause (if there are several ways to derive a clause). The usefulness information is used in the CrystalBall SAT solver [13]. First, a proof is generated without any clause deletion. Then, DRAT-trim identifies the usefulness of clauses. Then, criteria are derived to classify useful clauses. Finally, these new criteria are used in the CrystalBall SAT solver — although in that case, it did not improve the performance.

Own Contribution

Parallel Proofs [H8, 1]. Parallel SAT solvers can utilize clause sharing as an approach for parallel solving. This can occur between solver cores through clause exchange or by sharing clauses in memory. The former method is less intrusive, requiring only slight adjustments to top-performing solvers from previous competitions. It also enables easy strategy variation as different solvers implement distinct strategies. The latter method demands greater modifications to the SAT solver, as clause sharing prevents putting watched literals at the beginning due to different cores having varying watched literals.

Parallel SAT solvers can generate proofs, but there is no format such that each core produces it separately, especially with clause exchange in such proofs. This complication arises due to the need for correct addition and production ordering. Consequently, currently, the entire proofs must be within a single file, making it compatible with a single-core verifier.

The easiest approach is to let each SAT solver core produce its own proof and ensure that clause exchange happens only after the clause has been added to the proof (and not deleted yet). Physical clause sharing has an advantage: there is no need to repeat clauses for each shared clause as long as they are in a single memory location. This means much shorter proofs, as there is a single creation and a single deletion.

We experimented with proof checking of the certificates. We observed two interesting properties: constant size and checking efficiency. First, the proof size seems independent of the number of threads. This is an interesting observation because it means that the parallel cores do not exchange clauses that the other has already derived. This seems to indicate that the proof found by the SAT solver has a similar size, but the cores work on separate parts of the search space. We observed the same property for speed: the number of CPU cycles remained constant (so 2 cores are twice as fast as 1 core and half as fast as 4 cores). This metric is relevant for cloud-based solving where the user pays for CPU cycles.

Second, sharing in proofs is very important for efficiency of checking. This is not surprising: The checker has an active set of clauses. This set of clauses is the union of all active sets of all the cores (and the clauses whose deletion has been forgotten). So if each of the 32 cores has the same clause (which is very likely for short clauses like binary or ternary clauses), the checker will have it 32 times instead of 1 with physical sharing.

LRAT Proofs [H5, H4]. Directly checking proofs in DRAT format is costly, as it still requires searching (i.e., propagation, like in the SAT solver). To prevent repeated searches, verified checkers employ extended formats with information IDs for the propagation. They use DRAT-TRIM for CAKE_LPR or GRATGEN for GRAT to generate the (slightly different) enhanced formats. however, generating hints for the enhanced proof format can take longer than the solver's proof-finding time. A solution is to create the enriched proof format within the SAT solver itself.

A previous study resulted in the development of a new proof format [2], FRAT, which combines DRAT and hints. FRAT allows hints without requiring all of them, making it easier to develop the solver. However, the proof format is not fully compatible with DRAT when no hints are provided and requires a new translation tool for CAKE_LPR. Baek et al.'s implementation in CADICAL does not generate hints for all inprocessing techniques and does not support units in the input file.

We implement proof production and introduce hint generation for all techniques. Additionally, we properly fuzz our solver and support input of units. As a result, CADICAL generates proofs (both in binary and non-binary) with all hints. This has a performance cost, primarily due to writing large proofs to disk, as we have observed proofs exceeding 60 GB in size. We have observed one very lengthy proof checking (more than 24 h[1] hours) even when we generate all hints correctly and reported in issue 18 on GitHub.

The implementation requires always-activated clause IDs in CADICAL, but otherwise, our approach is no-cost and as compatible as possible (IDs slightly change the behavior of garbage collection as clauses are slightly larger) with the old behavior when LRAT proofs are not activated.

Overall, the entire workflow solving+trimming+checking is much faster than either using the FRAT checker or DRAT. It is the first time that a state-of-the-art solver is natively producing LRAT proofs.

Beyond SAT: Multipliers [H9]. Together with Kaufman and others, we developed a system requiring arithmetic proofs. To verify hardware-multiplier correctness, AMULET converts the circuit into a sequence of polynomials, representing each gate in the implementation. We use Gröbner bases for proof, as SAT solvers are insufficient for arithmetic. Kaufman implemented Gröbner bases transformation herself, because the existing ones are too slow for this application.

[1] https://github.com/digama0/frat/issues/18

We implemented two checkers, PACTRIM and PASTÈQUE, for verifying certificates. These checkers ensure the accuracy of each polynomial transformation. We did not need Gröbner basis verification for this task, only that the polynomial manipulations are correct. We did not find bugs in the rules we devised, but found some missing checks in the PACTRIM checker for the introduction of new variables.

The implementation of the checker was rather straightforward, thanks to the refinement approach, once we had sorted out how to make polynomials work. The Isabelle representation of polynomials is very general, but much of the low-level (like extracting a coefficient) did not work out automatically. However, SLEDGEHAMMER [16] was able to fill the holes most of the times. The refinement approach allowed us to use as concrete representation a list of coefficients and variables.

Future Research Topics

Certifying more systems is very important to ensure correctness. We are currently working on certificates for incremental SAT solving. A necessary first step before being able to verify a checker is to devise a proof format (and verify that it is useful).

One challenge is to make the entire certification process less "develop new tools for each application" and more "let's reuse the same building blocks." The Isabelle Archive of Formal Proof contains many existing data structures that can be useful, and the Isabelle Refinement Library can be used to replace abstract structures with concrete ones. However, not all data structures are associated with good imperative implementations, and memory handling can become very complicated. For PASTÈQUE, we ended up with string sharing, which required a refinement that basically added a mapping from integers to a string, with handling of out-of-memory in the mapping. It would be better to have an automated approach (at least for sharing without modifying a variable).

4 Verified SAT Solving

The highest level (Level 5) of trust we can reach is nowadays is fully verifying the SAT solver. This means that we verify that the given answer is correct for any input problem.

Two methods exist for verifying SAT solvers. The first, simpler approach verifies the UNSAT result by proving its correctness and simultaneously checks the model provided in the SAT result. Current approaches [17, 5] validate the derived learned clauses until the solver obtains the empty clause. However, this is partial verification, as the solver might not terminate or enter a loop. Essentially, the proof of correctness ensures that a proof checker could verify each step. Yet, it allows the solver to delete any clause, and it is essential to verify the model's correctness for the satisfiable answer. Programs resort to run-time checks, which are too challenging or not important enough to prove within their system.

The second approach is the one I used during my PhD: start with a formalization of CDCL and refine it towards code. The advantage is that this is an effort towards understanding the theory and the invariants of the code. Similar approaches exist but none has gone as far as generating code that solves problems of the SAT Competitions.

Different communities develop the two SAT solving approaches, leading to a different approach: partial verification is done bottom-up, starting from the code and checking that a specification is met with formal engines from Level 2, whereas full verification is done top-down, starting from CDCL down to code synthesis or extraction with interactive theorem provers. The former approach gives more control over the code (as it is written by the user). However, there is no deeper reason why bottom-up is not possible in Isabelle (for example, the tool AutoCorres imports C code in Isabelle).

Partial verification sometimes needs additional checks or work-arounds. For example, an invariant of clause learning is that the conflict is found as early as possible. In IsaSAT, we prove this by showing that propagation and conflict are eagerly found. Partial verification is not able to do that currently and instead either aborts or simply restarts the search ignoring the conflict. In the worst-case, this can lead to non-termination, but this does not harm

the partial verification. In IsaSAT, we actually need completeness to prove that we follow CDCL.

Full verification of an entire SAT solver is a long task, and we had to decide what techniques we want to represent. For example, our CDCL calculus does not support shrinking (from Chapter 2), because we did not consider it important enough yet to implement it. A technique with an even bigger impact on the code synthesis and invariants is chronological backtracking [14].

Surprisingly to us, no partial verification has ever verified the most important preprocessing technique, namely variable elimination: as it resolves clauses together, verification is simple. And fixing the assignment if the problem is satisfiable is done outside of the correctness property anyway.

We verified IsaSAT via a refinement approach: We start from a CDCL calculus and slowly refine the representation by replacing data structures (like multisets for clauses become lists) and changing the behavior (decision can take any literal in CDCL, and we replace this by a heuristic that gives a deterministic answer). At the last step, the Sepref tool replaces the functional data structure by imperative ones (like arrays instead of lists). Finally, the code is exported to Standard ML and we combine the resulting solver with a parser to run it on SAT problems.

One of the easier techniques to implement but hard to verify is the memory representation for clauses [12]. IsaSAT uses an arena: clauses (with the headers) are all allocated one after the other. This is very easy to do in a SAT solver as it simply requires to change the allocation strategy but no difference in usage. In the proof assistant, this is not possible, as we cannot control memory allocation. Therefore, we went for an explicit representation (the arena is really a big array of 32-bit words representing all clauses and headers).

Own Contributions

Inprocessing in IsaSAT [H3]. In order to improve the performance of IsaSAT, we made two major changes: (i) we modified the CDCL calculus to support simplification, and (ii) we changed the target code of the synthesis from Standard ML to LLVM IR (the language used by the Clang compiler, which is employed by Apple).

The first change introduced a new calculus called Pragmatic CDCL (PCDCL). At its core, it still runs CDCL but includes additional rules such as learning any non-tautological clause and dropping clauses. We also added the possibility to add non-implied clauses. We employed this technique for pure

literal deletion: when a literal appears only positively in the initial set of clauses, we can actually set it to true unconditionally. We also implemented clause deletion, which requires proving that we do not change the possible models.

Interestingly, the implementation of pure literal deletion resulted in invalid DRAT certificates. Although the technique is correct (we have proven this!), IsaSAT generates proofs (without any claims that the proofs are correct) to participate in the main track of the SAT Competition. However, the DRAT proof format does not distinguish between initial and learned clauses. As a consequence, the pure literal might not be pure with respect to all clauses, leading to incorrect proofs – even though IsaSAT provides the correct answer.

Example 4.1. *Consider the problem A, $\neg A$, $B \vee C$. The literal B appears only positively. Therefore, we can actually learn B (and therefore remove the clause $B \vee C$).*

However, the SAT solver could learn $\neg B \vee C$ (the problem is UNSAT). The literal B still appears only positively in the initial clauses, but it now also appears negatively in the learned clauses. DRAT checkers are not able to verify this addition of B and thus proof checking fails, even the addition does not change the status of the problem.

The "bug" from Example 4.1 was actually discovered during the SAT Competition. This is not a correctness issue of our solver, just a technique that we cannot use when generating DRAT proofs.

To validate that our PCDCL is useful, we implemented forward subsumption-resolution, that is, detecting when resolving two clauses yields a new shorter clause. Along the way, the solver must remember which clauses are redundant (and thus are removable) and which clauses are irredundant (must be kept for consistency).

Model Reconstruction In a master thesis I supervised, Katharina Wagener worked on model reconstruction: some simplification techniques like variable elimination used by SAT solvers subtly change the model, but there is an easy way to fix the model if there is one (no change is required if there is no solution). While this is verification Level 1, here Wagener did the work in Isabelle: This is the highest level of trust that is reachable at Level 1 – and this is the first step towards adding model reconstruction in IsaSAT.

The idea of reconstruction is that redundant clauses are removable, and fixing the model requires only flipping some literals from the explanation of why the clause was removed.

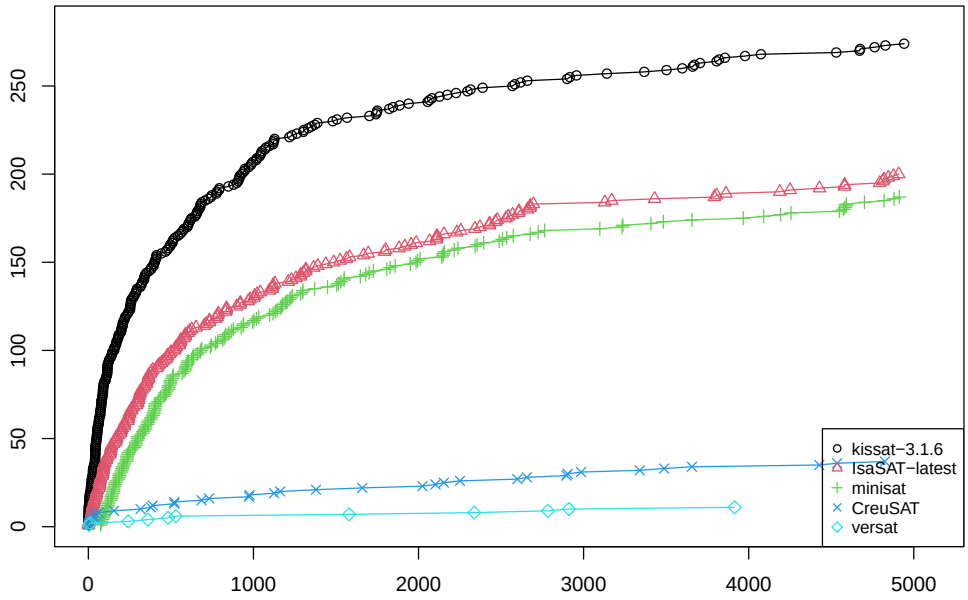


Figure 4.1: Performance comparison of SAT solvers as CDF of the SAT Competition 2022, with 7 GB RAM, 5000 s timeout (higher is better), with Kissat, Minisat 2006, the fully verified SAT solver IsaSAT, and the partially verified CreuSAT and `versat`.

During this work, we discovered an unexpected mismatch between the presentation on paper and the implementation claims. First, redundancy as defined by the paper is not the most general criterion. In particular, variable elimination does not work in all cases. There are three possible fixes: giving up on expressing variable elimination, changing the witness (in particular, this means that the implementation becomes more complicated), or restricting the construction to total models (over all variables that have ever been present in the formula). Fortunately, the implementation actually uses total models, so the implemented model reconstruction is correct (at least in theory).

Future Work

Verifying SAT solvers does not mean that we verify every aspect. In particular, for heuristics, there is no property on the detailed implementation. For example, we implement bumping to increase the importance of the variables

(important for the next decision) and we only bump variables from the formula, so the construction is correct. However, we do not verify that we bump the right variables. We recently discovered that we bumped the variables that are present in the conflict clause, while Cadical does not. Changing the implementation (which did not require changing any proofs), actually improved the performance, see Figure 4.1. This small change makes IsaSAT better than Minisat 2006 (with preprocessing).

We are currently working on including model reconstruction into IsaSAT. It turns out that we have to change our invariants slightly, in particular related to subsumption. With model reconstruction, the solution is actually better than the current theory, where our PCDCL keeps the clauses (but does not use them except for expressing invariants).

Since IsaSAT is the largest project where code generation is applied, it is exercising code generation of ITPs. Switching from SML generation to LLVM IR generation improved performance, but it came with a huge refactoring cost. In the future, we would like to improve the performance of the code generation (as the synthesis of some functions is actually taking several minutes), but also better understand if the code generation is missing any important optimizations. LLVM IR now supports code generation to parallel code, but it is currently not simple to use it for IsaSAT, because it is based on splitting an array into two, one part to each process.

5 Using SMT Proofs in Isabelle

Proving in interactive theorem provers, such as Isabelle or any other, is tedious because the built-in tactics are not very strong. To reduce the pain, it is possible to rely on external tools to provide a proof. This often happens through Sledgehammer [16]: It selects relevant theorems in Isabelle and translates them into the logic supported by some external automatic tools, before "importing" some information that the external tools give about the found proof. This is exactly the certification approach described in Chapter 3, where we use the proof instead of checking it.

Historically, one of the first approaches was to use external superposition solvers as "fact filters" and then use Isabelle's built-in ordered resolution Metis prover to refind the proof (and replay the proof through the Isabelle kernel to have a fully correct proof). This approach turned out not to be strong enough. Therefore, Isabelle nowadays uses the proof provided by the superposition solvers: It first tries if the UNSAT core (only the theorems needed to prove the goal) is sufficient for a built-in technique. If not, it redirects the proof to avoid a proof by contradiction, and it transforms then each step into a proof step in the built-in proof language Isar [16]. Checking each of those steps is simpler than checking the entire proof, which makes it more likely for a built-in tactic to solve the goal.

There is another community of people developing automated theorem provers, called SMT solvers. These solvers are based on a very different principle. They call a SAT solver (from Chapter 2) on an abstracted version of the input problem and slowly constrain the abstracted problem. The problem is constrained more until either a model of the abstracted problem that is also a model of the concrete problem or the abstracted problem is deemed unsatisfiable, meaning that the concrete problem is also unsatisfiable. For this, they have to guess instantiations of quantifiers. One simple example (with only linear arithmetic) is $x \leq 3 \vee x \geq 4$: The SAT solvers receive the abstraction $a \vee b$ without knowing what a and b mean. The SAT solver can return the model ab and the theory solver (here the arithmetic) refines the problem by adding the constraint that this model is not possible by adding $\neg a \vee \neg b$. This process continues until a valid model is found (here $x \leq 3$ and $\neg(x \geq 4)$).

SMT solvers all support problems in the format defined by the SMT-Lib (with some extensions). Many different logic fragments are defined and solvers can support only some of them. Problems generated by Isabelle heavily rely on quantifiers, while no array can be generated. There are three major SMT solvers cvc, veriT, and z3 supporting this fragment and able to generate proofs. The most active development currently is happening on CVC5, but VERIT has more readable and simpler code[1].

Proof generation is a much debated topic, because it is much more complicated than in the SAT case, where dumping clauses is sufficient as (nearly) all clauses are derived by resolution anyway. Z3 is generating proofs in a variant of natural deduction. Another suggested format for the proof solver SMT-INTERPOL is resolution-based [4]. The last proof format is called ALETHE (see below). It is very pragmatic: it makes it easy to generate proofs of transformation under quantifiers, thanks to its support of contexts. They make it possible to start a proof without knowing the result of the transformation yet. The drawback is that it is not easy to express global transformations like symmetry breaking.

Own Contribution

veriT Reconstruction [H11] and Alethe [8]. I have participated in the implementation of the reconstruction of VERIT proofs in SMT proofs. As mentioned, we check the proof written in the proof format Alethe instead of trusting it, unlike other approaches. I was also part of the proper definition of a semantics of the proof format [8].

The main motivation of this work is improving the success rate of the reconstruction. CVC4 supports techniques for instantiation that are not supported by Z3, making it impossible to use Isabelle's SMT tactic (as Z3 does not find a proof, there is nothing to reconstruct in Isabelle!). On the other hand, VERIT does support those techniques.

The work was at the intersection of working on VERIT and Isabelle. Therefore, we identified four strategies for VERIT in order to maximize performance on Isabelle problems. They mostly differ by the instantiation techniques used.

As we are working hand-in-hand with VERIT developers, they added information to the proof format, such as coefficients when proving unsatisfiability of a set of linear equations. The coefficients always exist due to the technique implemented in VERIT, and Isabelle's built-in strategy for linear arithmetic

[1] and I contributed one SAT technique to the built-in SAT solver, position saving.

is only able to reconstruct the steps when the coefficients are integers. Providing coefficients (as rational numbers with rounding) makes it possible for Isabelle to check the steps.

In order to improve the speed of the reconstruction, we *compressed* the proofs by removing some substeps and compressing some idioms. Most removed steps are actually renaming steps. In Alethe, transforming $\forall x.\ P\ x$ into $\forall veriT\_vr0.\ P\ veriT\_vr0$ involves:

1. under the assumption that $x = veriT\_vr0$, a proof that $P\ x = P\ veriT\_vr0$. This can actually involves multiple steps.
2. from Step 1 the conclusion that $(\forall x.\ P\ x) = (\forall veriT\_vr0.\ P\ veriT\_vr0)$
3. a logical tautology on equality $(\forall x.\ P\ x) = (\forall veriT\_vr0.\ P\ veriT\_vr0) \vee \neg(\forall x.\ P\ x) \vee (\forall veriT\_vr0.\ P\ veriT\_vr0)$
4. finally the conclusion that $\forall veriT\_vr0.\ P\ veriT\_vr0$ holds from the Steps 3 and 2 and the fact that $\forall x.\ P\ x$ holds.

However, in Isabelle names are just placeholder (only de Brujn indices are used in terms). Therefore, reflexivity of equality can already prove that $\forall x.\ P\ x$ and $\forall veriT\_vr0.\ P\ veriT\_vr0$ the same (as equality in Isabelle ignores the "name hints" given to the variables).

Overall, this work improves the success rate of replaying proof in Sledgehammer. It also improves the amount of time required to reconstruct the proof, because Sledgehammer can pick either the new veriT-based or the old Z3-based reconstruction method. In HOL-Library, even with the newly solved goals, the reconstruction is still faster than before.

IsaRARE [H1]. Since the last contribution, the SMT solver CVC4 [18] was renamed cvc5 [3] (with a different font). The work described below is done in cvc5.

One problem of SMT solvers is that it is hard to decide which rewrite rules to use. This is where RARE rules come in: the SMT solver cvc5 is parametrized over rules (at compile time), but they can easily be changed by recompiling the SMT solver to try new rules. Actually, the SMT solver applies all rules together and finds during proof production which rules have been applied – leading to possibly several rewriting steps in the final proof or none if the step is not necessary to derive $\bot$.

However, having many rules makes it complicated to ensure their correctness, especially since the semantics of the SMT-LIB is rather complicated.

Therefore, we developed a tool called IsaRARE to translate RARE rules to Isabelle. The rules are written to support all theories, including bit-vectors.

Lachnitt (partially under my supervision) developed the translation for IsaRARE. This is very complicated as the semantics of the SMT-LIB and of Isabelle differ. In particular, a lot of constraints in the SMT-LIB are basically dependent types, that Isabelle does not support: `extract i j b` extracts the bits from `j` to `i` in the bit-vector `b`, yielding a bit-vector of size `i-j+1`. In order to do this in Isabelle, we have to add type annotations. In particular, the definition of `concat` is highly complicated (and it is not possible to define for more than 2 arguments in Isabelle).

During the translation and the subsequent usage of Nitpick, several bugs were identified and fixed. However, not all rules have yet been proved – especially, the proofs can become very complicated over bit-vectors.

Future Research Topics

The logical next step is to fully support the reconstruction of cvc5 proofs. While veriT makes it possible to find many proofs already, they are still proofs that veriT is not able to find fast enough. Therefore, this would improve the reconstruction rate. Another open question is how stable proofs are: currently, Isabelle relies on a rather old Z3 version. While it would be nice to update it, because it now supports more features, there are two issues:

1. the proof output is not maintained and we discovered that some proofs contain extra variables.[2]
2. the proof format has slightly changed since in an undocumented way.

Updating Z3 would give an idea if the tools are still able to find proofs again (as they depend on very few facts usually). Moreover, it remains to be seen how many proof formats will be produced in the next year. In particular, there is ongoing work for a new proof format for Z3 that will be more DRAT-like (in particular, no dependencies between steps are indicated, requiring to trim the proofs). The aim of the main Z3 developer is to make it easier to produce proofs. However, this goes at the cost of consuming the proof to reconstruct it.

The Alethe proof format is also expanding with fixes and with additional information to make reconstruction easier. For example, type information is

[2] https://github.com/Z3Prover/z3/issues/5073

part of the anchor steps instead of requiring to parse until the conclusion of the step (which can be arbitrary further away). However, Isabelle is currently not using that fact and still parses the entire proof before reconstructing it.

6 Conclusion

In the last chapters, I have encapsulated my contributions to the field of automated reasoning. These contributions can be categorized into four areas, aligned with the different levels of verification. My first contribution involves enhancing base tools, specifically SAT solvers. Beyond contributing to a better understanding of the evolution since the 1990s and highlighting that improved performance is not solely due to different benchmarks, I have also worked on improving their efficiency. This was achieved by optimizing the main CDCL loop's performance through altered search directions and extracting information from CNF to identify definitions.

My second contribution focuses on boosting trust in these tools. I contributed to improving the proof format generated by SAT solvers, demonstrating the possibility of creating a proof containing all the necessary hints for verified proof checkers with minimal overhead. Although extending this to the state-of-the-art SAT solver Kissat remains unclear, we have produced proofs for the parallel SAT solver Gimsatul. Proofs make it possible to extract information on the search, and we found that proof length remained roughly constant, regardless of thread number. However, proof checking times proved substantial. Additionally, I worked on certifying proofs beyond SAT with Pastèque.

My third contribution consists of developing the only verified SAT solver, IsaSAT. As the only fully verified solver capable of competing in the main track of the SAT Competition alongside non-verified solvers (even if it still finishes last or second to last), it achieves the highest level of trust currently possible. The significant performance gap observed since the previous version from my PhD thesis can be mostly attributed to the porting to Isabelle_LLVM (enabling the use of the superior Clang compiler) and (a little to) the addition of inprocessing[1].

My final contribution centers around proof usage, particularly when verifying IsaSAT. I aim to use external tools such as SMT solvers without blindly trusting them. Therefore, Isabelle checks the correctness of the dumped

[1]According to discussion with Armin Biere, forward subsumption seems to be mostly important when variable elimination is done, which we left as future work.

proofs, improving efficiency and the number of checked proofs, thereby reducing human interaction as each failed attempt now requires less manual intervention for completion. I have also worked on defining a proper standard for the proof format and am currently collaborating with Hanna Lachnitt on the reconstruction in CVC5, including machine words (bitvectors).

Bibliography of Submitted Scientific Publications

[H1] Hanna Lachnitt, Mathias Fleury, Leni Aniva, Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. "IsaRare: Automatic Verification of SMT Rewrites in Isabelle/HOL". In: *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*. Ed. by Bernd Finkbeiner and Laura Kovács. Vol. 14570. Lecture Notes in Computer Science. Accepted at TACAS 2024, I supervised Hanna on her work in Isabelle. Springer, 2024, pp. 311–330.

[H2] Armin Biere, Fleury, Mathias, Nils Froleyks, and Marijn Heule. "The SAT Museum". In: *Pragmatics of SAT 2023*. Ed. by Matti Järvisalo and Daniel Le Berre. I wrote the first version of the paper and did some of the experiments and porting. 2023.

[H3] Fleury, Mathias and Peter Lammich. "A more Pragmatic CDCL for IsaSAT and targetting LLVM". In: *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*. Ed. by Brigitte Pientka and Cesare Tinelli. I did most of the formalization and the paper and Peter helped in the initial porting. Springer, 2023, pp. 207–219. URL: https://m-fleury.github.io/ox-hugo/FleuryLammich-CADE29.pdf.

[H4] Florian Pollitt, Fleury, Mathias, and Armin Biere. "Efficient Proof Checking with LRAT in CaDiCaL (work in progress)". In: *24th GMM/ITG/GI Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems, MBMV 2023, Freiburg, Germany, March 23-23, 2023*. Ed. by Armin Biere and Daniel Große. I supervised Florian and wrote parts of the paper. VDE, 2023, pp. 64–67.

[H5] Florian Pollitt, Fleury, Mathias, and Armin Biere. "Faster LRAT Checking than Solving with CaDiCaL". In: *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July*

4-8, 2023, Alghero, Italy. Ed. by Luca Pulina, Laura Pandolfo, and Dario Guidotti. Vol. 271. LIPIcs. I supervised Florian and wrote parts of the paper. 2023, 21:1–21:12.

[H6] Shaowei Cai, Xindi Zhang, Fleury, Mathias, and Armin Biere. "Better Decision Heuristics in CDCL through Local Search and Target Phases". In: *J. Artif. Intell. Res.* 74 (2022). I did the implementation in Glucose, the experiments with CaDiCaL, the analysis, and helped in the design in CaDiCaL/Kissat., pp. 1515–1563.

[H7] Fleury, Mathias and Armin Biere. "Mining definitions in Kissat with Kittens". In: vol. 60. 3. I did the experiments, the analysis, and helped in the design. 2022, pp. 381–404.

[H8] Fleury, Mathias and Armin Biere. *Scalable Proof Producing Multi-Threaded SAT Solving with Gimsatul through Sharing instead of Copying Clauses*. Ed. by Matti Järvisalo and Daniel Le Berre. I helped in the design and wrote parts of the paper. 2022. URL: https://arxiv.org/abs/2207.13577.

[H9] Daniela Kaufmann, Fleury, Mathias, Armin Biere, and Manuel Kauers. "Practical Algebraic Calculus and Nullstellensatz with the Checkers Pacheck and Pastèque and Nuss-Checker". In: *Formal Methods in System Design*. I did the entire work Isabelle and the details of the side-conditions where done together to find out why I was having issues in the proof. 2022.

[H10] Fleury, Mathias and Armin Biere. "Efficient All-UIP Learned Clause Minimization". In: *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, SAT 2021, Barcelona, Spain, July 5-9, 2021, Proceedings*. Ed. by Chu Min Li and Felip Manyà. Vol. 12831. LNCS. I wrote parts of the paper, found the proof for minimization, and implemented shrinking in one solver. Springer, 2021, pp. 171–187.

[H11] Hans-Jörg Schurr, Fleury, Mathias, and Martin Desharnais. "Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant". In: *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*. Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. LNCS. I did the entire work Isabelle and most of the experiments and large parts of the writing. Springer, 2021, pp. 450–467.

Bibliography

[1] Armin Biere, Fleury, Mathias, and Florian Pollitt. "CaDiCaL_vivinst, IsaSAT, Gimsatul, Kissat, and TabularaSAT Entering the SAT Competition 2023". In: *Proc. of SAT Competition 2023 – Solver and Benchmark Descriptions*. Ed. by Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2023-1. Department of Computer Science Report Series B. University of Helsinki, 2023, pp. 14–15.

[2] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. "A Flexible Proof Format for SAT Solver-Elaborator Communication". In: *Log. Methods Comput. Sci.* 18.2 (2022).

[3] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442.

[4] Jochen Hoenicke and Tanja Schindler. "A Simple Proof Format for SMT". In: *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022*. Ed. by David Déharbe and Antti E. J. Hyvärinen. Vol. 3185. CEUR Workshop Proceedings. CEUR-WS.org, 2022, pp. 54–70. URL: https://ceur-ws.org/Vol-3185/paper9527.pdf.

[5] Sarek Høverstad Skotåm. "CreuSAT, Using Rust and Creusot to create the world's fastest deductively verified SAT solver". Master's

Thesis. University of Oslo, 2022. URL: https://www.duo.uio.no/handle/10852/96757.

[6] Shaowei Cai and Xindi Zhang. "Deep Cooperation of CDCL and Local Search for SAT". In: *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*. Ed. by Chu-Min Li and Felip Manyà. Vol. 12831. Lecture Notes in Computer Science. Springer, 2021, pp. 64–81.

[7] Muhammad Osama, Anton Wijs, and Armin Biere. "SAT Solving with GPU Accelerated Inprocessing". In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12651. Lecture Notes in Computer Science. Springer, 2021, pp. 133–151. URL: https://doi.org/10.1007/978-3-030-72016-2%5C_8.

[8] Hans-Jörg Schurr, Fleury, Mathias, Haniel Barbosa, and Pascal Fontaine. "Alethe: Towards a Generic SMT Proof Format (extended abstract)". In: *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, Pittsburg, USA, 11th July 2021*. Ed. by Chantal Keller and Fleury, Mathias. Vol. 336. Electronic Proceedings Ain Theoretical Computer Science. Open Publishing Association, 2021, pp. 49–54.

[9] Armin Biere and Fleury, Mathias. "Chasing Target Phases". In: *Pragmatics of SAT 2020*. Ed. by Matti Järvisalo and Daniel Le Berre. 2020.

[10] Nick Feng and Fahiem 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*. Ed. by Luca Pulina and Martina Seidl. Vol. 12178. Lecture Notes in Computer Science. Springer, 2020, pp. 28–45.

[11] Johannes Klaus Fichte, Markus Hecher, and Stefan Szeider. "A Time Leap Challenge for SAT-Solving". In: *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*. Ed. by Helmut Simonis. Vol. 12333. Lecture Notes in Computer Science. Springer, 2020, pp. 267–285.

[12] Fleury, Mathias. "Optimizing a Verified SAT Solver". In: *NASA Formal Methods—11th International Symposium, NFM 2019, Houston, TX, USA, May 7–9, 2019, Proceedings*. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Vol. 11460. Cham: Springer, 2019, pp. 148–165. ISBN: 978-3-030-20652-9.

[13] Mate Soos, Raghav Kulkarni, and Kuldeep S. Meel. "CrystalBall: Gazing in the Black Box of SAT Solving". In: *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*. Ed. by Mikolás Janota and Inês Lynce. Vol. 11628. Lecture Notes in Computer Science. Springer, 2019, pp. 371–387.

[14] Alexander Nadel and Vadim Ryvchin. "Chronological Backtracking". In: *Theory and Applications of Satisfiability Testing—SAT 2018—21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. Ed. by Olaf Beyersdorff and Christoph M. Wintersteiger. Vol. 10929. Lecture Notes in Computer Science. Springer, 2018, pp. 111–121.

[15] Armin Biere. "CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017". In: *SAT Competition 2017: Solver and Benchmark Descriptions*. Ed. by Tomáš Balyo, Marijn J. H. Heule, and Matti Järvisalo. Vol. B-2017-1. Department of Computer Science Series of Publications B. University of Helsinki, 2017, pp. 14–15. URL: http://fmv.jku.at/papers/Biere-SAT-Competition-2017-solvers.pdf.

[16] Jasmin Christian Blanchette, Sascha Böhme, Fleury, Mathias, Steffen Juilf Smolka, and Albert Steckermeier. "Semi-intelligible Isar Proofs from Machine-Generated Proofs". English. In: *Journal of Automated Reasoning* (2015), pp. 1–46. ISSN: 0168-7433.

[17] Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. "versat: A Verified Modern SAT Solver". In: *Lecture Notes in Computer Science*. Vol. 7148. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 363–378. ISBN: 9783642279393.

[18] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *Computer Aided Verification—23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177.

[19] Robert Brummayer, Florian Lonsing, and Armin Biere. "Automated Testing and Debugging of SAT and QBF Solvers". In: *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. Ed. by Ofer Strichman and Stefan Szeider. Vol. 6175. Lecture Notes in Computer Science. Springer, 2010, pp. 44–57. URL: https://doi.org/10.1007/978-3-642-14186-7%5C_6.

[20] Gilles Audemard and Laurent Simon. "Predicting Learnt Clauses Quality in Modern SAT Solvers". In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. Ed. by Craig Boutilier. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 399–404. URL: http://ijcai.org/Proceedings/09/Papers/074.pdf.

[21] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. ISBN: 978-1-58603-929-5.

[22] Niklas Sörensson and Armin Biere. "Minimizing Learned Clauses". In: *Theory and Applications of Satisfiability Testing—SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30—July 3, 2009. Proceedings*. Ed. by Oliver Kullmann. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009, pp. 237–243.

[23] HyoJung Han and Fabio Somenzi. "Alembic: An Efficient Algorithm for CNF Preprocessing". In: *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*. IEEE, 2007, pp. 582–587. URL: https://doi.org/10.1145/1278480.1278628.

[24] Niklas Eén and Niklas 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*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer, 2003, pp. 502–518.

[25] João P. Marques Silva and K.A. Sakallah. "GRASP—A new search algorithm for satisfiability". In: *Proceedings of International Conference on Computer Aided Design*. IEEE Comput. Soc. Press, 1996, pp. 220–227. ISBN: 0818675977.

[26] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782.

[27] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (1960), pp. 201–215.

Part II

Submitted Scientific Publications

7 Understanding and Improving SAT Solvers

Conference Paper

[1] Fleury, Mathias and Armin Biere. "Efficient All-UIP Learned Clause Minimization". In: *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, SAT 2021, Barcelona, Spain, July 5-9, 2021, Proceedings*. Ed. by Chu Min Li and Felip Manyà. Vol. 12831. LNCS. I wrote parts of the paper, found the proof for minimization, and implemented shrinking in one solver. Springer, 2021, pp. 171–187.

Efficient All-UIP Learned Clause Minimization

Mathias Fleury and Armin Biere

Johannes Kepler University, Linz, Austria
{armin.biere, mathias.fleury}@jku.at

Abstract. In 2020 Feng & Bacchus revisited variants of the all-UIP learning strategy, which considerably improved performance of their version of CaDiCaL submitted to the SAT Competition 2020, particularly on large planning instances. We improve on their algorithm by tightly integrating this idea with learned clause minimization. This yields a clean shrinking algorithm with complexity linear in the size of the implication graph. It is fast enough to unconditionally shrink learned clauses until completion. We further define trail redundancy and show that our version of shrinking removes all redundant literals. Independent experiments with the three SAT solvers CaDiCaL, Kissat, and Satch confirm the effectiveness of our approach.

1 Introduction

Learned clause minimization [18] is a standard feature in modern SAT solvers. It allows to learn shorter clauses which not only reduces memory usage but arguably also helps to prune the search space. However, completeness of minimization was never formalized nor proven. Using Horn SAT [9] we define trail redundancy through entailment with respect to the reasons in the trail and show that the standard minimization algorithm removes all redundant literals (Sect. 2).

Minimization, in its original form [18], only removes literals from the initial *deduced clause* during conflict analysis, i.e., the 1st-unique-implication-point clause [21]. In 2020 Feng & Bacchus [11] revisited the *all-UIP* heuristics with the goal to reduce the size of the deduced clause even further by allowing to add new literals. In this paper we call such advanced minimization techniques *shrinking*. In order to avoid spending too much time in such shrinking procedures the authors of [11] had to limit its effectiveness. They also described and implemented several variants in the SAT solver CaDiCaL [2]. One variant was winning the planning track of the SAT Competition 2020. The benchmarks in this track require to learn clauses with many literals on each decision level.

As Feng & Bacchus [11] consider minimization and all-UIP shrinking separately, they apply minimization first, then all-UIP shrinking, and finally again minimization (depending on the deployed strategy/variant), while we integrate both techniques into one simple algorithm. In contrast, their variants process literals of the deduced clause from highest to lowest decision level and eagerly introduce literals on lower levels. Thus their approach has to be guarded against actually producing larger clauses and can not be run unconditionally (Sect. 3).

We integrate minimization and shrinking in one procedure with linear complexity in the size of the implication graph (Sect. 4). Processing literals of the deduced clause from lowest to highest level allows us to reuse the minimization cache, without compromising on completeness, thus making it possible to run the shrinking algorithm unconditionally until completion. On the theoretical side we prove that our form of shrinking fulfills the trail redundancy criteria.

Experiments with our SAT solvers Kissat, CaDiCaL, and Satch show the effectiveness of our approach and all-UIP shrinking in general. Shrinking decreases the number of learned literals, particularly on the recent planning track. We also study the amount of time used by the different parts of the transformation from a conflicting clause to the shrunken learned clause (Sect. 5).

Regarding related work we refer to the *Handbook of Satisfiability* [7], particularly for an introduction to CDCL [17], the main algorithm used by state-of-the-art SAT solvers. This work is based on the classical minimization algorithm [18], which Van Gelder [19] improved by making it linear (in the number of literals) in the implication graph without changing the resulting minimized clause. The original all-UIP scheme [21] was never considered to be efficient enough to be part of SAT solvers, until the work by Feng & Bacchus [11]. We refer to their work for a detailed discussion on all-UIPs. Note that, Feng & Bacchus [11] consider their algorithm to be independent of minimization, more like a post-processing step, while we combine shrinking and minimization for improved efficiency. The technical report with the proofs of all theorems is available [13].

2 Minimization

We first present a formalization of what minimization actually achieves through the notion of "trail redundancy". Then the classical deduced clause minimization algorithm is revisited. It identifies literals that are removable and others literals called *poison* that are not. The algorithm uses a syntactic criterion, but removes exactly the trail redundant literals. We present five existing criteria to detect (ir)redundancy earlier and prove their correctness.

When a SAT solver identifies a conflicting clause, i.e., a clause in which all literals are assigned to false, it analyzes the clause and first deduces a 1st-unique-implication-point clause [17,21]. This *deduced clause* is the starting point for minimization and shrinking. The goal is to reduce the size of this clause by removing as many literals as possible. The following redundancy criterion specifies if a literal is removable from the deduced clause.

Definition 1 (Semantic Trail Redundancy). *Given the formula F_M composed only of the reason annotating propagated literals in the trail M and the conflicting clause D such that $M \vDash \neg D$. The literal $L \in \neg M$ is called* redundant *iff* $F_M \vDash \neg L \vee (D \setminus \{L\})$.

For this definition we only consider redundancy with respect to the reasons in the trail (ignoring other clauses in the formula). Note that, most SAT solvers only use the first clause in the watch lists to propagate, even though "better"

clauses might trigger the same propagation. For instance PRECOSAT scans watch lists to find such cases [3]. However, due to potential cyclic dependencies, deducing the shortest learned clause is difficult [20].

Theorem 2 (Redundant Literals are Removable). *If $L \vee D$ is the deduced clause and L is redundant, then D is conflicting and entailed.*

Our next theorem states that the order of removal does not impact the outcome and that it is possible to cache whether a literal is (ir)redundant.

Theorem 3. *Literals stay (ir)redundant after removal of redundant literals.*

The reason $L \vee C$ annotates the propagation literal $L^{L \vee C}$ in the trail. Minimizing the deduced clause consists in recursively resolving with the reasons: If the clause becomes smaller, it is used. Duplicate literals are removed from the clause. Algorithm 1 shows a recursive implementation that resolve away the literal L without addition of literals. The minimization algorithm applies to every conflicting clause but is only applied to the *deduced clause* [21], namely the deduced clause after the first unique implication point was derived.

The minimization algorithm is standard in SAT solvers with several improvements. First, they use efficient data structures to efficiently check if a literal is in the deduced clause. Second, they use caching: if a literal was deemed (un)removable before, the same outcome is used again. Caching successes and failures [19] make the algorithm linear in the size of the implication graph. Literals that can not be removed are called *poison*.

Our definition of trail redundancy is semantic, while the minimization algorithm uses relies on syntactic criteria to determine if a literal is removable or not. We show that both criteria are equivalent by using a result of Horn satisfiability.

Definition 4 (Transition System by Dowling and Gallier [9]). *Consider the following rewriting system defined for Horn formulas, starting from the start symbol I*

1. *For every clause $L \vee \neg L_1 \vee \cdots \vee \neg L_n$, we consider the associated rewrite rule $\neg L \to \neg L_1 \cdots \neg L_n$ (where n can be zero).*
2. *For every clause $\neg L_1 \vee \cdots \vee \neg L_n$, we consider the rewrite rule $I \to \neg L_1 \cdots \neg L_n$.*

In Definition 4, given our SAT context the step $\neg L_1 \cdots \neg L_n$, represents the entailed clause $\neg L_1 \vee \cdots \vee \neg L_n$. One rewriting step is a resolution step.

Theorem 5 (Dowling and Gallier [9]). *Given a satisfiable Horn formula, a literal is true iff it can be rewritten to $\bot$.*

The transition system from Definition 4 is not linear. As far we are aware, this is the first description of minimization algorithm in terms of Horn SAT.

Theorem 6. *Algorithm 1 is the same as the transition system from Definition 4.*

Function `IsLiteralRedundant`(L, d, C)

Input: Literal L assigned to *true*, recursion depth d, deduced clause C
Output: Whether L can be removed

if L is a decision **then**
  return *false*
$D \vee L \longleftarrow$ reason(L);
foreach *literal* $K \in D$ **do**
  if $\neg$`IsLiteralRedundant`($\neg K$, $d+1$, C) **then**
    return *false*
return *true*

Function `MinimizeSlice`(B, C)

Input: A clause C (passed by reference) and a subset B of C to minimize
Output: The minimized clause with redundant literals in B removed

foreach $K \in B$ **do**
  $R \longleftarrow \emptyset$
  if `IsLiteralRedundant`($\neg K$, 0, C) **then**
    $R \longleftarrow R \cup \{K\}$
$C \longleftarrow C \backslash R$

Algorithm 1: Basic recursive minimization algorithm similar to [18].

Theorem 7 (Equivalence Syntactic and Semantic Redundancy). *Both notions of redundancy are equivalent. In particular, every redundant literal is also removable.*

In our formalization of learned clause minimization for our verified SAT solver IsaSAT [12], we use a different definition of redundancy, namely $F_M \vDash \neg L \vee D_{<_M L}$ where $D_{<_M L}$ are all the literals of D that appear before L in the trail M. This definition is equivalent but it makes more explicit that only literals that appear before L are relevant. We have not formalized completeness while working on IsaSAT since we only cared about correctness.

Theorem 8. *A literal L is redundant iff $F_M \vDash \neg L \vee D_{<_M L}$.*

Our implementation relies on the alternative definition: It sorts the literals in the clause by its position on the trail. Each literal, starting from the lowest position, is checked. If it is not redundant, it is marked as present in the deduced clause for efficient checking. This reduces the number of flags (like testing if a literal is present in the deduced clause) to reset. Instead we could use d: When $d = 0$, the condition "L is in the deduced clause" does not apply.

Thanks to caching both successes and failures, the complexity is linear in the number of literals of the trail. Compared to our simple break conditions, more advanced criteria are possible.

Theorem 9 (Poison Criteria).

1. *If a literal appears on the trail before any other literal of the deduced clause on a decision level, then it is not redundant.*

Function `IsLiteralRedundantEfficient`(L, d, C)

Input: Literal L assigned to *true*, recursion depth d, deduced clause C
Output: Whether L can be removed

if status of L is cached in minimization cache **then**
  return cached value

if any advanced poison criterion from Theorem 9 applies (uses d) **then**
  return *false*

if L is root-level assigned (unit) or $\neg L \in C$ **then**
  return *true*

if L is a decision **then**
  return *false*

$D \vee L \longleftarrow$ reason(L)
foreach $K \in D$ **do**
  if $\neg$`IsLiteralRedundantEfficient`($\neg K$, $d+1$, C) **then**
    Cache *false* for L
    return *false*
Cache *true* for L
return *true*

Algorithm 2: Advanced minimization algorithm equivalent to Algorithm 1.

2. *Literals with a decision level not in the deduced clause are not redundant.*
3. *Literals that are alone on a given decision level are not redundant (Knuth).*

The proof relies on the fact that the SAT solver propagates literals eagerly. This is not the case globally if the SAT solver uses chronological backtracking [15, 16] but remains correct for the reason clauses. The second and third point are widely used (e.g., in MiniSAT and Glucose), whereas the first one is a novelty of CaDiCaL and is not described so far. Root-level assigned false literals can also appear in deduced clauses and be removed without recursing over their reasons.

Theorem 10. *Literals at level 0 are redundant.*

Algorithm 2 combines the two ideas that are described here, the caching and the advanced poison criteria. The ideas 1 and 3 from Theorem 9 require data structures that are not present in every SAT solver, namely the position τ of each literal in the trail. Doing so was not necessary until now, but it is required for shrinking. In our solvers, we also use the depth to limit the number of the recursive calls and avoid stack overflows. The implementation in MiniSAT [10] (and all derived solvers like Glucose [1]) uses a non-recursive version, but it requires two functions, one for depth zero and another for the recursive case.

3 Shrinking

After detecting conflicting clauses, the SAT solver analyzes them and deduces the first unique-implication point or 1-UIP [7], where only one literal remains

Function `MinAllUIPShrinkSlice`(B, C)

Input: Slice B of literals of the deduced clause C on the (slice) level
Output: B unchanged or shrunken if *min-alluip* is successful

$E \longleftarrow \emptyset$
while $|B| > 1$ **do**
  Remove from B last assigned literal $\neg L$
  $D \vee L \longleftarrow \text{reason}(L)$
  if $\exists K \in D \backslash C$ assigned at lower level not already in C **then**
    $E \longleftarrow E \cup \{L\}$
  else
    $B \longleftarrow B \cup \{K \in D \mid K \text{ assigned on slice level}\}$
Replace in deduced clause C original B with $B \cup E$

Function `MinAllUipShrinking`(C)

Input: The deduced clause C (passed by reference)
Output: The shrunken clause using the *min-alluip* strategy

$C' \longleftarrow C$
foreach Level i of literals in the deduced clause – highest to lowest **do**
  $B \longleftarrow \{L \in C \mid L \text{ assigned at level } i\}$
  `MinAllUIPShrinkSlice`(B, C)
Replace C with saved original deduced clause C' unless $|C| < |C'|$

Algorithm 3: Shrinking algorithm *min-alluip* from Feng&Bacchus [11].

on the current (largest) decision level. This is the first point where the clause is propagating, fixing the current search direction. The idea of 1-UIP can be applied on every level in order to produce shorter clauses. We call this process *shrinking*. It differs from minimization because it adds new literals to the deduced clause.

If fully applied, shrinking derives a subset of the decision-only clause. Therefore, it is limited. Feng & Bacchus [11] (abbreviated F&B from now on) have used various heuristics like not adding literals of low importance, without a clear winner across all implementations. We focus on their *min-alluip* variant. It applies the 1-UIP on every level. For each literal in the clause, the solver resolves with its reason unless a literals from a new level is added, thus making sure that the LBD or "glue" [1] is not increased, an important metric, which seems to relate well to the "quality" of learned clauses. In their implementation, if the clause becomes longer, the minimized clause would be used instead.

Algorithm 3 shows the implementation of *min-alluip*. It considers the set of all literals of the deduced clause on the same level, or *slice* (same as a block if no chronological backtracking [15,16] is allowed). Each slice is shrunken starting from the highest level. It resolves each literal of the slice with its reason or fails when adding new literals on lower levels. Because SAT solvers propagate eagerly, $|B| \geq 1$ is an invariant of the while loop (and L cannot be a decision literal).

The key difference between shrinking and minimization is that reaching the UIP is a *global* property, namely of all literals on a level, and not of a single

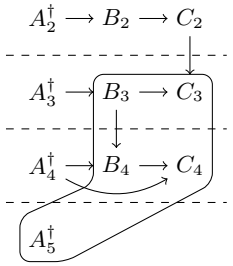


Fig. 1. Conflict example

literal. This means that testing redundancy is a depth-first search algorithm while shrinking is a breadth-first search algorithm on the implication graph.

Example 11. Consider the implication graph from Figure 1. The algorithm starts with the highest level, namely with B_4 and A_4. The level is reduced to A_4 introducing the already present B_3. On the next level, C_3 cannot be removed because it would import level 2. The resulting clause $\neg A_5 \vee \neg A_4 \vee \neg A_3$ is smaller and is used instead of the original clause.

F&B unfortunately do not provide source code nor binaries used in their experiments. Therefore we focus on their version of CaDiCaL submitted [14] to the SAT Competition 2020. It implements only one of their strategies, which, as far we can tell, matches the variant *min-alluip* [11] described above, while code for the other variants is incomplete or missing.

4 Minimizing and Shrinking

In contrast to F&B our algorithm minimizes literal slices of the deduced clause assigned on a certain level starting from the lowest to highest level. This enables us to remove all redundant literals on-the-fly. After presenting our algorithm we study its complexity and then discuss its implementation in our SAT solvers CaDiCaL, Kissat, and Satch.

The main loop of our Algorithm 4 interleaves shrinking and (if shrinking failed) minimization. For each slice of literals in the deduced clauses assigned on a certain level we then attempt to reach the 1-UIP, similarly to Algorithm 3. If this fails, we minimize the slice. This also allows to lift some restriction on shrinking: only non-redundant literals interrupt the search for the 1-UIP. We start from the lowest level to keep completeness of minimization.

Example 12. Consider the implication graph from Figure 1. The algorithm starts with the slice of literals on the lowest decision level, namely with B_3 and C_3. No UIP can be found because it would import level 2. Level 1 is shrunken to A_4. The shrunken clause is $\neg A_5 \vee \neg A_4 \vee \neg B_3 \vee C_3$

Function `ShrinkingSlice(`B, C`)`

Input: Slice B of literals of the deduced clause C on a single (slice) level
Output: B unchanged or shrunken to UIP if our new method is successful

while $|B| > 1$ **do**
  Remove from B last assigned literal $\neg L$
  $D \vee L \longleftarrow \text{reason}(L)$
  if $\exists\, K \in D \backslash C$ at lower level and $\neg$`IsLiteralRedundant(`$\neg K$, 1, C`)`
  then
    return with failure (keep original B in C)
  else
    $B \longleftarrow B \cup \{K \in D \mid K \text{ on slice level}\}$
Replace in deduced clause C original B with the remaining UIP in B

Function `Shrinking(`C`)`

Input: The deduced clause C (passed by reference)
Output: The shrunken and minimized clause using our new strategy

foreach Level i of literals in the deduced clause – lowest to highest **do**
  $B \longleftarrow \{L \in C \mid L \text{ assigned at level } i\}$
  `ShrinkingSlice(`B, C`)`
  if shrinking the slice failed **then** `MinimizeSlice(`B, C`)`;

Algorithm 4: Our new method for integrated shrinking with minimization.

As mentioned before, for efficiency a cache is maintained during minimization to know whether a literal is redundant or not.

Theorem 13 (Shrinking and Redundancy). *Redundant literals remain redundant during shrinking.*

Theorem 13 ignores irredundant literals because new literals are added to the deduced clause, allowing for more removable literals. This explains why F&B propose (in one variant of shrinking) to minimize again after shrinking. For the same reason we do not check if literals are redundant on the current level, since added literals (e.g., new 1st UIPs) invalidate the literals marked as "poisoned". Instead, we check for redundancy of literals on lower levels and on current level only after shrinking them, when the literals on the slice level are fixed.

Example 14 (Minimization during shrinking). Consider the following trail

$$A_1^\dagger B_1^{B_1 \vee \neg A_1} \quad A_2^\dagger B_2^{B_2 \vee \neg B_1 \vee \neg A_2} \quad A_3^\dagger$$

where † marks a decision and the deduced clause is $\neg A_1 \vee \neg B_2 \vee \neg A_3$. Shrinking cannot remove B_2 because it would introduce the new literal B_1 on lower levels, unless it is determined to actually be redundant (A_1 is in the deduced clause).

To keep the complexity linear, when interleaving minimization with shrinking as shown in Algorithm 4, we maintain a global shared minimization cache, not

reset between minimizing different slices. A more complicated solution consists in minimizing up-front (as in the implementation of F&B in [14]), followed by shrinking, and if shrinking succeeds, reset the poison literals on the current level.

Resetting only literals on the current level is important for reducing the runtime complexity from quadratic to linear in the size of the implication graph. As we are shrinking "in order" (from lowest to highest decision level) we can keep cached poisoned (and removable) literals from previous levels, thus matching the overall linear complexity of (advanced) minimization.

Our solution also avoids updating the minimization cache more than once during shrinking. When a slice is successfully reduced to a single literal, all shrunken literals are marked as redundant in the minimization cache. The process is complete in the sense that no redundant literals remain.

Theorem 15 (Completeness). *All redundant literals are removed.*

This result relies on the fact that during the outer loop no literal on a lower level is added to the deduced clause. If this would be allowed (as in Algorithm 3), the poisoned flag has to be reset and minimization redone, yielding a quadratic algorithm. However, the theorem says nothing about minimality of the shrunken clause if we allow to add new literals, as in the following example.

Example 16 (Smaller Deduced Clause). Consider the trail

$$A_1^\dagger B_1^{B_1 \vee \neg A_1} C_1^{C_1 \vee \neg B_1} \quad A_2^\dagger B_2^{B_2 \vee \neg A_2} C_2^{C_2 \vee \neg B_2 \vee \neg B_1} \quad A_3^\dagger$$

and the deduced clause $\neg C_1 \vee \neg B_2 \vee \neg C_2 \vee \neg A_3$. The clause is neither minimized nor shrunken by our algorithm, but can be shrunken to the smaller $\neg B_1 \vee \neg B_2 \vee \neg A_3$.

In Algorithm 4, on the one hand, shrinking could use a priority queue (implemented as binary heap) to determine the last assigned literal in B. Then for each slice, we have a complexity of $\mathcal{O}(n_b \log n_b)$ for shrinking where n_b is the number of literals at the slice level in the implication graph. On the other hand, minimization of all slices is linear in the size of the implication graph. Overall the complexity is $\mathcal{O}(\text{glue} \cdot n \log n)$ where the "glue" is the number of different slices (and a number that SAT solvers try to reduce heuristically) and n the maximum of the n_b. However, note that, bumping heuristics require sorting of the involved literals anyhow either implicitly or explicitly [6].

Instead of representing the slice B as a priority queue, implemented as binary heap, to iterate over its literals, it is also possible to iterate over the trail directly as it is common in conflict analysis to deduce the 1st-UIP clause. Without chronological backtracking, the slices on the trail are disjoint and iterating over the trail is efficient and gives linear complexity $\mathcal{O}(|\text{glue}| \times |\text{max\_trail\_slice\_length}|)$, i.e., linear in the size of the implication graph.

With chronological backtracking slices on the trail are not guaranteed to be disjoint. Therefore, in the worst case, iterating over a slice along the trail might require to iterate over the complete trail. In principle, this could give a quadratic

complexity for chronological backtracking without using a priority queue for B. In our experiments both variants produced almost identical run-times and thus we argue that the simpler variant of going over the trail should be preferred.

We have implemented the algorithm from the previous section in our SAT solvers CaDiCaL [5], Kissat [5], and Satch [4]. The implementation is part of our latest release in the file `shrink.c` (`shrink.cpp` for CaDiCaL).[1] Note that, Satch is a simple implementation of the CDCL loop with restarts and was written to explain CDCL. It does not feature any in- nor preprocessing yet.

We either traverse the trail directly or use a radix heap [8] as priority queue. Unlike the implementation by F&B, our priority queue contains only the literals from the current slice until either shrinking fails or the 1-UIP is found. It allows for efficient popping and pushing trail positions. Note that, radix heaps require popped elements to be strictly decreasing, and as the analysis follows reverse trail order, we first compute the maximum trail position of literals in the considered slice and then index literals by their offsets on the tail from this maximum trail position. The literal position in the trail is not cached in every SAT solver, but was already maintained in Kissat and CaDiCaL.

5 Experiments

We have implemented our algorithm in the SAT solvers CaDiCaL, Kissat (the winner of the SAT Competition 2020), and Satch and evaluated them on benchmark instances from the SAT Competition 2020 on an 8-core Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (turbo-mode disabled). For both tracks we used a memory limit of 128 GB (as in the SAT Competition 2020). We tested 3 configurations, shrink (shrinking and minimizing), minimize, and no-minimize (neither shrinking nor minimizing). Due to space constraint we only give graphs for some solvers but findings are consistent across all of them.

Tables 1 for Kissat and Satch show that minimization is more important than shrinking, but the latter still improves performance for Kissat. In the planning track, running time decreases significantly, whereas the impact on the main track is smaller. Compared to the main track, the planning problems require much more memory and memory usage drops substantially with shrinking. For Satch, we observe a slight performance decrease. Figures 2 and 3 show that even if shrinking solves only a few more problems, the speedup is significant.

In all our SAT solvers we distinguish between *focused mode* (many restarts) and *stable mode* (few restarts). Note that CaDiCaL uses the number of conflicts to switch between these modes which is rather imprecise: in stable mode decision frequency is lower while the conflicts frequency is higher compared to focused mode and accordingly the fraction of running time spent in conflict analysis and thus minimization and shrinking increases in stable mode compared to focused mode. To improve precision both Kissat and Satch measure the time by estimating the number of possible cache misses instead, called "ticks" [5]. By default

[1] Source code and log files are available at `http://fmv.jku.at/sat_shrinking`.

Table 1. Results for new solvers on the SAT Competition 2020 benchmarks

Solver	Track	Configuration	Solved	PAR-2	Average clause size
Kissat	Main Track (400 problems)	shrink	**270**	**1561735**	**46**
		minimize	267	1 566 688	110
		no-minimize	235	1 891 872	183
	Planning Track (200 problems)	shrink	**85**	**1197799**	**5 398**
		minimize	83	1 222 535	13 076
		no-minimize	74	1 325 957	16 637
Satch	Main Track (400 problems)	shrink	196	2 271 119	**46**
		minimize	**203**	**2240351**	144
		no-minimize	159	2 621 070	370
	Planning Track (200 problems)	shrink	**85**	**1212977**	**5 043**
		minimize	80	1 250 861	11 854
		no-minimize	72	1 338 592	15 474
CaDiCaL 1.4.0	Main Track (400 problems)	shrink	**240**	**1 870 484**	**90**
		minimize	233	1 939 998	121
		no-minimize	194	2 280 897	153
	Planning Track (200 problems)	shrink	**73**	**1 334 718**	**4 885**
		minimize	64	1 454 186	7 799
		no-minimize	42	1 615 676	11 767

Kissat also counts the number of such ticks during shrinking and minimization. To avoid the bias introduced by this technique in terms of influencing mode switching we deactivated this feature in our experiments (only for Kissat).

We analyzed the results on the main track in more details over all instances (i.e., until timeout or memory out), not only over solved instances. The amount of time (in percentage of the total) more than doubles when activating shrinking: it goes from 6.3 % to 14.3 % of the total amount of time (Figure 5). However, the size of the clauses is reduced with a similar ratio: It drops from 110 to 46 (183 without minimization). On the planning track, it drops from 13 076 to 5 398 literals on average (16 637 without minimization).

To compare our method to the *min-alluip* implementation, which is based on CaDiCaL 1.2.1, we backported our *shrinking* algorithm to CaDiCaL 1.2.1 too. The results are in Table. 2. The only difference is the shrinking algorithm, hence there are not differences for the minimize and no-minimize configuration. The F&B version performs slightly better than our version. An interesting observation is that CaDiCaL 1.2.1 learns much larger clauses than Kissat and Satch but also larger than the latest CaDiCaL version. The effect can be partially explained by the stable mode that is much longer than on the other solvers. We have also experimented with minimizing separately from shrinking instead of combining them. As long as the cache is shared there is very little performance difference. Figure 7 shows the CDF for the main track.

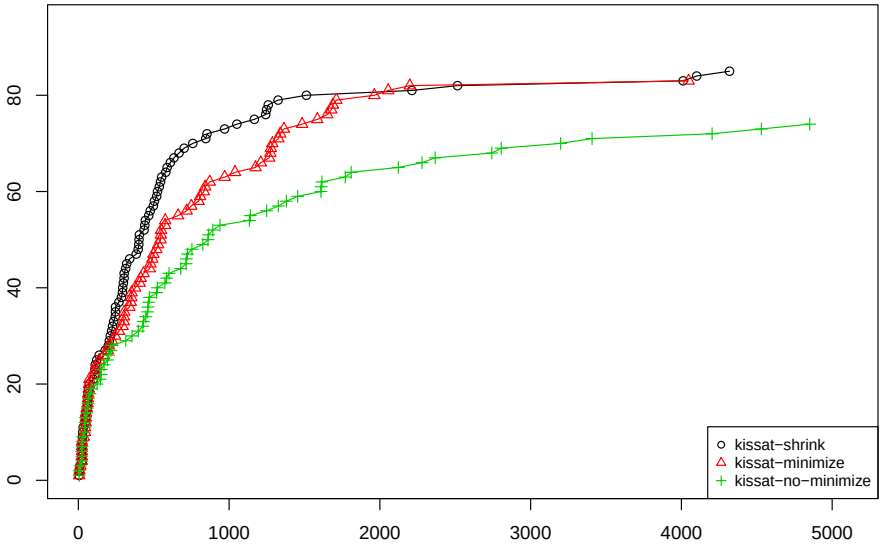


Fig. 2. KISSAT solving time on the planning track.

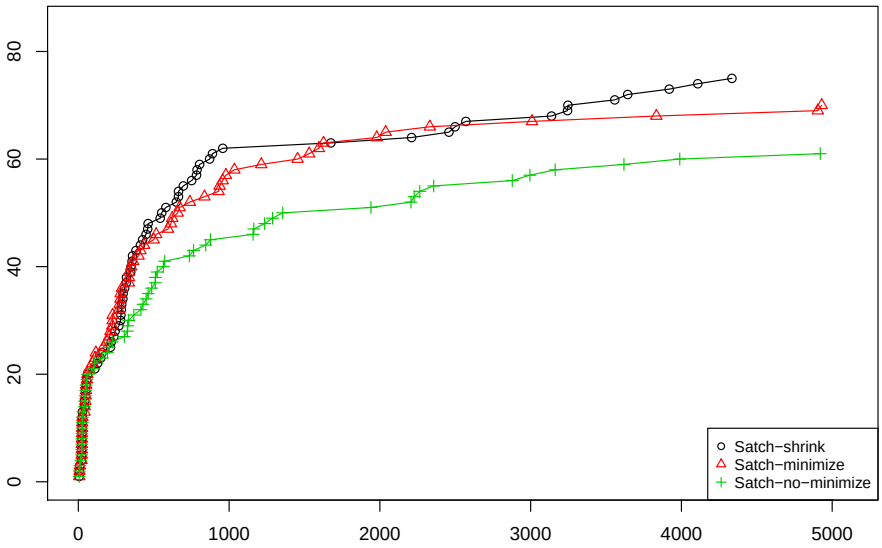


Fig. 3. SATCH solving time on the planning track.

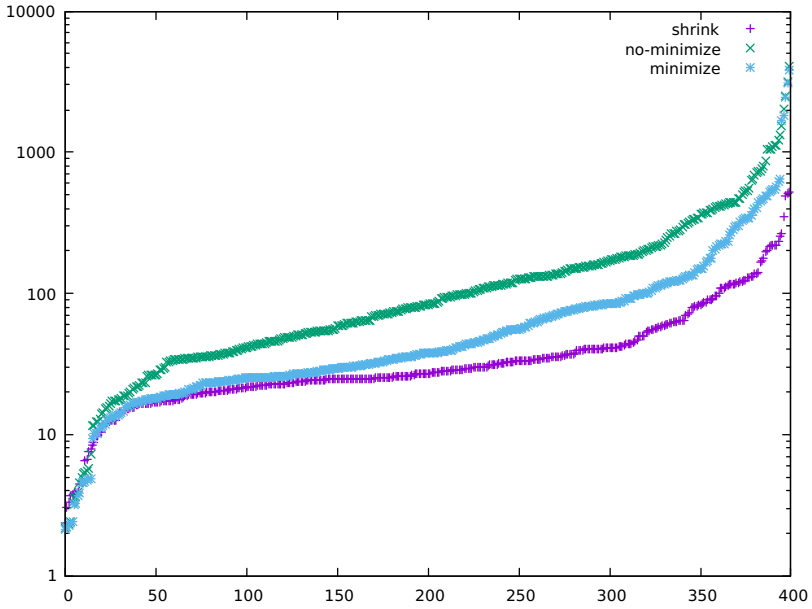


Fig. 4. Absolute sizes of learned clauses of KISSAT on main track.

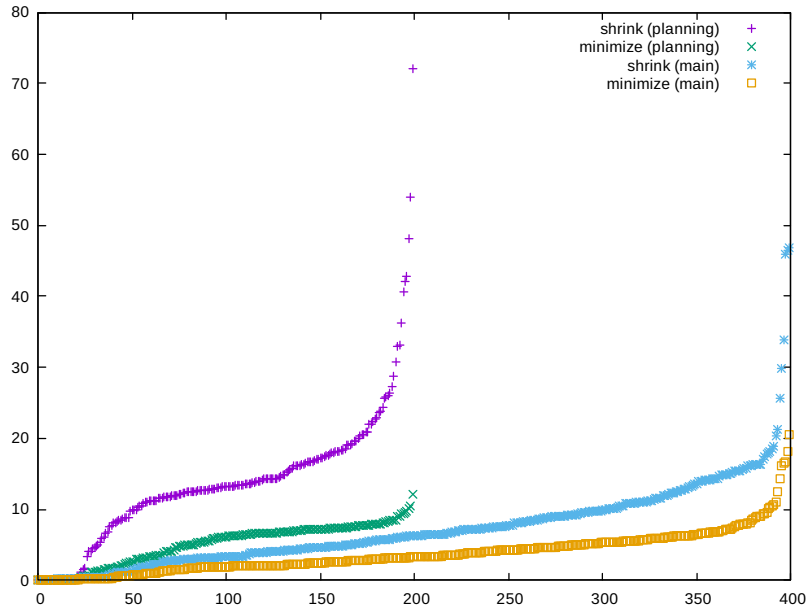


Fig. 5. Amount of time in percent spent during shrinking and minimization of KISSAT.

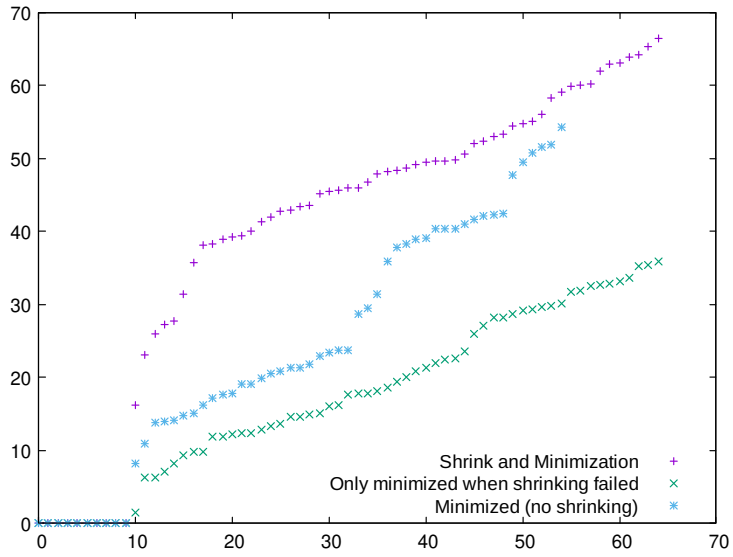


Fig. 6. Percentage removed literals in learned clauses for CaDiCaL in planning track.

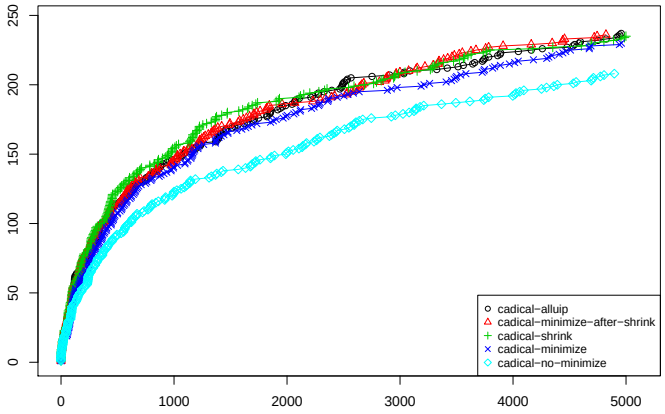


Fig. 7. Comparison between CaDiCaL-1.2.1 with shrinking (this paper) and the F&B version on the main track.

Table 2. Results for solvers based on CaDiCaL 1.2.1 on the SAT Competition 2020 benchmarks with a memory limit of 128 GB, following the SAT Competition

Solver	Track	Configuration	Solved	PAR-2	Average clause size
shrinking (this paper)	Main Track (400 problems)	shrink	235	**1 897 387**	**92**
		minimize	230	1 972 949	135
		no-minimize	208	2 184 920	187
	Planning Track (200 problems)	shrink	73	1 351 542	5 373
		minimize	63	1 454 871	6 433
		no-minimize	39	1 643 665	9 874
min-alluip [11, 14]	Main Track	shrink	**237**	1 904 745	104
	Planning Track	shrink	**81**	**1 271 930**	**3 261**

Figure 6 shows percentages of removable literals on the planning track. Shrinking removes more literals than the subsequent minimization (and more than minimization alone).

We have mentioned the complexity difference between using a radix heap and iterating over the trail. We have implemented both versions in our three SAT solvers. We compare both version but could not observe any significant difference. We believe that this is due to the fact that finding the next literal is actually very efficient: it is in the trail (that is in cache anyways) and we check a single flag. We attempted to force the worst case by enforcing chronological backtracking, but performance remained similar.

6 Conclusion

We presented a simple linear algorithm which integrates minimization and shrinking and is guaranteed to remove all redundant literals. In practice it can be run to completion unconditionally. Our implementation and evaluation with several SAT solvers show the benefit of our approach and confirm effectiveness of shrinking in general.

An open question is how to extend our notion of trail redundancy to capture that new literals can be added in order to reduce size. This would allow to formulate completeness of shrinking in the same way as we did for minimization.

Acknowledgment

This work is supported by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), and the LIT AI Lab funded by the State of Upper Austria. We also thank Sibylle Möhle and the anonymous reviewers for suggesting textual improvements.

References

1. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: IJCAI. pp. 399–404 (2009), http://ijcai.org/Proceedings/09/Papers/074.pdf
2. Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT Competition 2018. In: Heule, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2018-1, pp. 13–14. University of Helsinki (2018)
3. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Tech. rep., FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University (aug 2021)
4. Biere, A.: The SAT solver Satch. Git repository (2021), https://github.com/arminbiere/satch, last Accessed: March 2021
5. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
6. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Heule, M., Weaver, S.A. (eds.) SAT 2015. LNCS, vol. 9340, pp. 405–422. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4_29
7. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
8. Cherkassky, B.V., Goldberg, A.V., Silverstein, C.: Buckets, heaps, lists, and monotone priority queues. SIAM J. Comput. **28**(4), 1326–1346 (1999). https://doi.org/10.1137/S0097539796313490
9. Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae. J. Log. Program. **1**(3), 267–284 (1984). https://doi.org/10.1016/0743-1066(84)90014-1
10. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. LNCS, vol. 2919, pp. 502–518. Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37
11. Feng, N., Bacchus, F.: Clause size reduction with all-UIP learning. In: SAT. LNCS, vol. 12178, pp. 28–45. Springer (2020). https://doi.org/10.1007/978-3-030-51825-7_3
12. Fleury, M.: Formalization of logical calculi in Isabelle/HOL. Ph.D. thesis, Saarland University, Saarbrücken, Germany (2020), https://tel.archives-ouvertes.fr/tel-02963301
13. Fleury, M., Biere, A.: Efficient all-UIP learned clause minimization (extended version). Tech. Rep. 21/3, Johannes Kepler University Linz, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2021). https://doi.org/10.350/fmvtr.2021-3, https://doi.org/10.350/fmvtr.2021-3
14. Hickey, R., Feng, N., Bacchus, F.: Cadical-trail, Cadical-alluip, Cadical-alluip-trail and Maple-LCM-Dist-alluip-trail at the SAT Competition. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, p. 10. University of Helsinki (2020)

15. Möhle, S., Biere, A.: Backing backtracking. In: SAT. LNCS, vol. 11628, pp. 250–266. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_18
16. Nadel, A., Ryvchin, V.: Chronological backtracking. In: SAT. LNCS, vol. 10929, pp. 111–121. Springer (2018). https://doi.org/10.1007/978-3-319-94144-8_7
17. Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 131–153. IOS Press (2009). https://doi.org/10.3233/978-1-58603-929-5-131
18. Sörensson, N., Biere, A.: Minimizing learned clauses. In: SAT. LNCS, vol. 5584, pp. 237–243. Springer (2009). https://doi.org/10.1007/978-3-642-02777-2_23
19. Van Gelder, A.: Improved conflict-clause minimization leads to improved propositional proof traces. In: SAT. LNCS, vol. 5584, pp. 141–146. Springer (2009). https://doi.org/10.1007/978-3-642-02777-2_15
20. Van Gelder, A.: Generalized conflict-clause strengthening for satisfiability solvers. In: SAT. Lecture Notes in Computer Science, vol. 6695, pp. 329–342. Springer (2011). https://doi.org/10.1007/978-3-642-21581-0_26
21. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in Boolean satisfiability solver. In: ICCAD. pp. 279–285. IEEE Computer Society (2001). https://doi.org/10.1109/ICCAD.2001.968634

Conference Paper

[1] Fleury, Mathias and Armin Biere. “Mining definitions in Kissat with Kittens”. In: vol. 60. 3. I did the experiments, the analysis, and helped in the design. 2022, pp. 381–404.

Noname manuscript No.
(will be inserted by the editor)

Mining Definitions in Kissat with Kittens

Mathias Fleury · Armin Biere

the date of receipt and acceptance should be inserted later

Abstract Bounded variable elimination is one of the most important preprocessing techniques in SAT solving. It benefits from discovering functional dependencies in the form of definitions encoded in the CNF. While the common approach pioneered in SatELite relies on syntactic pattern matching, our new approach uses cores produced by an embedded SAT solver, Kitten. In contrast to a similar semantic technique implemented in Lingeling based on BDD algorithms to generate irredundant CNFs, our new approach is able to generate DRAT proofs. We further discuss design choices for our embedded SAT solver Kitten. Experiments with Kissat show the effectiveness of this approach.

1 Dedication

We dedicate this rather technical SAT paper to the memory of Ed Clarke. He was one of the first to see the tremendous potential of SAT solving not only in model checking, but more general in verification and beyond. His vision to use SAT for model checking, the encouragement and guidance he gave to two Post-Docs working on this topic (the 2nd author and Yunshan Zhu), which then lead to our multiple awards winning joint work on Bounded Model Checking [5–9, 16], clearly plays a pivotal role in the history of the SAT revolution we are witnessing today.

Bounded Model Checking turned out not only to become the first practical application of SAT but also, even though highly debated initially, lead to a paradigm shift in using formal verification, trading completeness for scalability. This controversy can also be seen as the starting point of other highly-influential work in the model checking community, particularly Ken McMilan's work on interpolation [30] and then the development of the IC3 algorithm by Aaron Bradley [14], which both also rely on SAT solving but try to keep completeness without sacrifying scalability too much.

Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria
Chair of Computer Architecture, Albert-Ludwigs-University, Freiburg, Germany
E-mail: fleury@informatik.uni-freiburg.de and E-mail: biere@informatik.uni-freiburg.de

This success of SAT in model checking motivated new research on SAT solving, including the seminal work at Princeton yielding the Chaff [32] SAT solver, which is standing on the shoulders of another seminal work around the Grasp solver from Michigan [36], and also turbo-charged the use of decision procedures originating in the automated theorem proving community in the form of SMT. This SAT revolution is a corner stone of the more broader adoption of automated reasoning in many applications, from classical hardware to software verification as well as scheduling cloud jobs. We believe without Ed this would not have happened.

2 Introduction

Preprocessing and particularly inprocessing [26] is a key feature of modern SAT solvers, the latter being part of every winner of the SAT competition since 2013. Arguably the most important pre- and inprocessing technique is bounded variable elimination (BVE). Even though in its unbounded form, elimination is a decision procedure for SAT, in the context of preprocessing bounded variable it is not run until completion. The idea of BVE is to iteratively eliminate one variable from the problem by resolving every occurrence away without adding redundant clauses. Furthermore, the difference between the number of added and removed clauses is bounded in practical implementation (Section 3).

Definability is a concept that reduces the number of clauses to add. It consists in recognizing a definition of x such that $x \leftrightarrow f(a_1, \ldots, a_n)$ from the input formula in conjunctive normal form (CNF). The simplest example are gates like $x \leftrightarrow a_1 \wedge a_2$ that can be efficiently detected. Detecting gates reduces the number of resolvents because not all clauses have to be resolved together. A simple approach is to syntactically recognize gates as encoded in the CNF input. This approach is for example used in CaDiCaL [10] and CryptoMiniSat [39].

This syntactic approach (Section 4) is limited though and fails to recognize "irregular" gates not characterized by a simple gate type (such as And gates). It also fails to detect gates after elimination of one of the input variables. Recently semantic approaches based on Padoa's theorem [34] have been developed with applications in model counting [28] and a similar technique exists for (D)QBF reasoning [35, 37]. In both approaches a SAT solver is used as oracle to find gate clauses. In this paper we follow this line of research and extend our SAT solver Kissat [12] to detect gates semantically. It uses a simple SAT solver called Kitten, called as an oracle to find gate clauses (Section 5). Our definition of gate detection is equivalent to previous approaches, even though our method never explicitly reconstructs the function (Section 6).

Our technique discovers gates but it does not need to know which are the inputs (Section 7). One interesting property about gates is that we do not need to resolve gate clauses among themselves. However, this only holds if the full clause is found and not a subset of the clause. If those clauses are forgotten, an unsatisfiable problem can become satisfiable (Section 8). Syntactic detection of gates is faster and detects most useful gates. So Kissat first finds gates syntactically and then calls Kitten to find other gates semantically (Section 9).

It turns out that the performance of the sub-solver Kitten has a non-negligible impact on the overall performance, as it is frequently called to find definitions with different environment clauses in which a candidate variable to be eliminated

occurs. Basically KITTEN is a very simple CDCL solver with watched literals but for instance without blocking literals. A key feature of KITTEN for semantic gate detection is that it can be "cleared" efficiently avoiding reallocation of internal data structures (Section 10). It further can be instructed to keep antecedents of learned clauses in memory and thus can compute clausal cores in memory.

Experiments on benchmarks from the SAT Competition 2020 show that our new elimination method has only a minor impact on performance and runtime, but it does eliminate substantially more variables, even after syntactic extraction is employed first. Thus definition extraction is effective (Section 11).

We finish with related work (Section 12). The idea of not generating redundant unnecessary clauses relates to blocked clause elimination (BCE), a simplification technique that can remove clauses. Iser [24] also used a SAT solver in the context of gate identification, but he does not use it to identify a gate, but only to check "right uniqueness" of already identified set of clauses.

This paper is an substantially extended version of our very brief presentation in the system description [12] of KISSAT from the SAT Competition 2021 and an extension of our (unpublished) *Pragmatics of SAT Workshop 2021 (POS'21)* presentation [11]. Compared to the system description, we have significantly extended all explanations and give more details about KITTEN. Last but not least we report detailed experiments.

3 Bounded Variable Elimination

In principle, eliminating variables from a formula reduces the search space in solving the formula exponentially with the number of removed variables. However, this argument is only sound as long the formula does not increase in size geometrically with the number of eliminated variables. Otherwise we would have found a procedure to polynomially solve SAT.

Thus the basic idea of bounded variable elimination is to only eliminate variables in a formula, for which the resulting formula is not bigger than the original formula, i.e., where the size increase due to variable elimination is bounded. This procedure can be implemented efficiently and in practice is considered the most effective preprocessing technique, particularly for industrial instances.

The basic approach works as follows. Let x be a variable considered to be eliminated from the CNF F. We split F syntactically into three parts

$$F = \underbrace{F_x \wedge F_{\bar{x}}}_{E(F,x)} \wedge \Delta(F,x),$$

where F_ℓ is the CNF of clauses of F which contain literal ℓ, with $\ell \in \{x, \bar{x}\}$ and $\Delta(F,x)$ contains the remaining clauses without x nor $\bar{x}$. We call $E(F,x) = (F_x \wedge F_{\bar{x}})$ the *environment* of x. As usual tautologies do not have to be considered, where a clause is called *tautological* or *trivial* if it contains a variable x and its negation $\bar{x}$.

Let x be a variable and H_x and $H_{\bar{x}}$ CNFs where clauses in H_ℓ all contain ℓ, we define the[1] *set of resolvents* of H_x and $H_{\bar{x}}$ over x as follows:

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

[1] If two clauses can be resolved over two different variables, the resulting resolvents are tautological. Thus the resolution operator "$\otimes$" does not really need to be parameterized by x.

As usual we interpret a CNF also as a set of clauses. The goal of variable elimination is to resolve all clauses of $F_{\bar{x}}$ with all clauses of F_x and replace $E(F,x)$ with the obtained resolvents, that is replacing the formula F by $(F_x \otimes F_{\bar{x}}) \wedge \Delta(F,x)$.

The process described so far is just a reformulation of "clause distribution" from the original DP procedure [17]. What turns it into the most important preprocessing techniques of today's SAT solvers is the idea of eliminating a variable if the difference between the number of added (resolvent) clauses and removed clauses (containing the eliminated variable x) is bounded [2, 3, 18, 40]. There are various possibilities to set this bound, and even increase it dynamically [33], which are orthogonal to the discussion of this paper.

Enforcing that the size of the formula does not grow too much during variable elimination restricts the number of variables that can be eliminated and thus the effectiveness of variable elimination. It is therefore beneficial to determine whether certain resolvents are redundant, i.e., implied by the resulting formula, and do not need to be added. This will allow additional variables to be eliminated, for which the size limit is hit without considering redundant resolvents.

Finally, as the elimination of a variable produces a formula which is satisfiability equivalent but not logically equivalent to the original formula (unless the formula is unsatisfiable), we need a way to reconstruct models of the original formula given a model of the simplified formula. This can be achieved by saving the eliminated clauses on a "reconstruction stack" and the interested reader might want to consult [13, 21, 26] for further details.

4 Gate Extraction

Already when introducing the SatELite preprocessor [18], it was proposed to extract subsets of "gate clauses" from F_x and $F_{\bar{x}}$ that 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 the solver to ignore resolvents between non-gate clauses (since those are implied). Assume that F can be decomposed as follows

$$F \;\equiv\; \overbrace{G_x \wedge H_x}^{F_x} \wedge \overbrace{G_{\bar{x}} \wedge H_{\bar{x}}}^{F_{\bar{x}}} \wedge \Delta(F,x)$$

where $G \equiv G_x \wedge G_{\bar{x}}$ are the *gate clauses*, i.e., the Tseitin encoding of a circuit gate with output x, H_x and $H_{\bar{x}}$ the remaining *non-gate clauses* of F containing x and $\bar{x}$ respectively, and $\Delta(F,x)$ the remaining clauses without x nor $\bar{x}$. The original technique from SatELite [18] would then use

$$F \;\equiv\; (F_x \otimes F_{\bar{x}}) \wedge \Delta(F,x) \;\equiv\; (G_x \otimes H_{\bar{x}}) \wedge (G_{\bar{x}} \otimes H_x) \wedge \Delta(F,x)$$

and only consider the smaller set of resolvents on the right, as both $G_x \otimes G_{\bar{x}}$ as well $H_x \otimes H_{\bar{x}}$ can be omitted from $F_x \otimes F_{\bar{x}}$, even though the former are tautological resolvents and thus ignored anyhow. To give a concrete example consider the following formula containing three gate clauses, encoding an And gate $x = a \wedge b$, and four non-gate clauses.

$$F = \underbrace{(\bar{a} \vee \bar{b} \vee x)}_{G_x} \wedge \underbrace{(a \vee \bar{x}) \wedge (b \vee \bar{x})}_{G_{\bar{x}}} \wedge \overbrace{(c \vee x) \wedge (d \vee x)}^{H_x} \wedge \overbrace{(e \vee \bar{x}) \wedge (f \vee \bar{x})}^{H_{\bar{x}}} \wedge \underbrace{(\bar{c} \vee \bar{d} \vee \bar{e} \vee \bar{f})}_{\Delta(F,x)}$$

Resolving all clauses with x or $\bar{x}$ results in the following CNF.

$$
\begin{array}{rl}
F'' \equiv & \\
\cancel{(\bar{a} \vee \bar{b} \vee a) \wedge (\bar{a} \vee \bar{b} \vee b)} \wedge & \text{tautological } G_x \otimes G_{\bar{x}} \text{ resolvents} \\
(\bar{a} \vee \bar{b} \vee e) \wedge (\bar{a} \vee \bar{b} \vee f) \wedge & \text{kept } G_x \otimes H_{\bar{x}} \text{ resolvents} \\
(a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d) \wedge & \text{kept } G_{\bar{x}} \otimes H_x \text{ resolvents} \\
(c \vee e) \wedge (c \vee f) \wedge (d \vee e) \wedge (d \vee f) \wedge & \text{redundant } H_x \otimes H_{\bar{x}} \text{ resolvents} \\
(\bar{c} \vee \bar{d} \vee \bar{e} \vee \bar{f}) & \text{kept } \Delta(F, x)
\end{array}
$$

Eliminating x in the original CNF F of 8 clauses results in CNF F'' with 13 clauses in total, but includes 2 tautological clauses, thus actually only has 11 non-tautological clauses. Without further ignoring the 4 redundant resolvents in $H_x \otimes H_{\bar{x}}$ bounded variable elimination (even up to allowing for introducing two more clauses) would still not eliminate x. If the AND gate is detected and non-gate clauses are not resolved against non-gate clauses, we end up with 7 clauses and x is eliminated.

Finding such gate clauses was originally based on syntactic pattern matching, by in essence trying to invert the Tseitin encoding. This is best explained for AND gates. Given an elimination candidate x and $\ell \in \{x, \bar{x}\}$. We go over all "base clauses" $C = (\ell \vee \ell_1 \vee \cdots \vee \ell_n)$ and check whether F also contains all $(\bar{\ell} \vee \bar{\ell}_i)$ for $i = 1 \ldots n$. If this is the case, we found the n-ary AND gate $\ell = (\bar{\ell}_1 \wedge \cdots \wedge \bar{\ell}_n)$ with gate clauses $G_\ell = \{C\}$ and $G_{\bar{\ell}} = \{(\bar{\ell} \vee \bar{\ell}_i) \mid i = 1 \ldots n\}$. If $\ell = x$ then x is the output of an AND gate. If $\ell = \bar{x}$, then x is the output of an OR gate $x = (\ell_1 \vee \cdots \vee \ell_n)$. For the special case $n = 1$ this amounts to extracting bi-implications (equivalences). According to our benchmarks (Section 11), extracting AND gates this way already gives the largest benefit but similar syntactical extraction techniques exist for XOR or IFTHENELSE gates.

Detecting gates syntactically, however, is not very robust and our SAT solver LINGELING [4] implements a very different technique inspired by BDD algorithms. It converts the environment clauses into a BDD (actually a function table), eliminates variables there, and translates the result back to a CNF using Minato's algorithm [19, 31], which produces a redundancy-free CNF. More details are provided in the preprocessing chapter of the 2nd edition of the Handbook of SAT [13].

Figure 1 shows a CDF of the number of solved instances of the last LINGELING release with and without this technique. On these problems from the SAT Competition 2020, deactivating this technique (`smallve0`) gives better performance. Remember that LINGELING is not developed anymore and was not trained on competition problems since 2016. Figure 2 gives the amount of time spent during variable elimination. As LINGELING's semantic variable elimination algorithm is arguably too costly, we take this as an additional motivation to look into different algorithms for semantic gate detection. The second issue with the implementation is that it cannot produce a DRAT proof of the transformation.

5 Definition Mining With a SAT Solver

Instead of only syntactically extracting definitions, our new version of KISSAT tries to extract gate clauses semantically by checking satisfiability of the conjunction

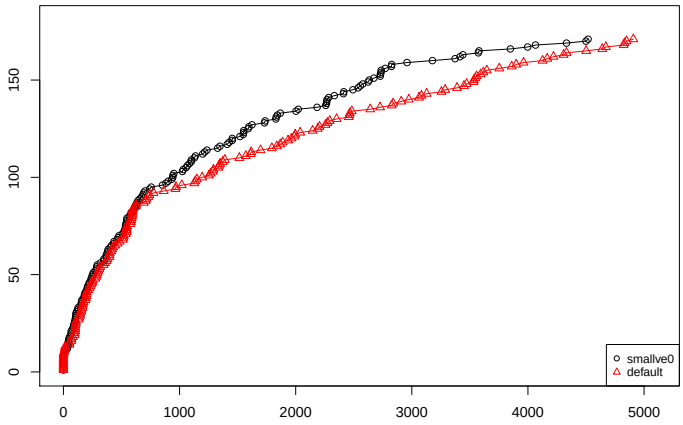


Fig. 1 Lingeling with and without variable elimination

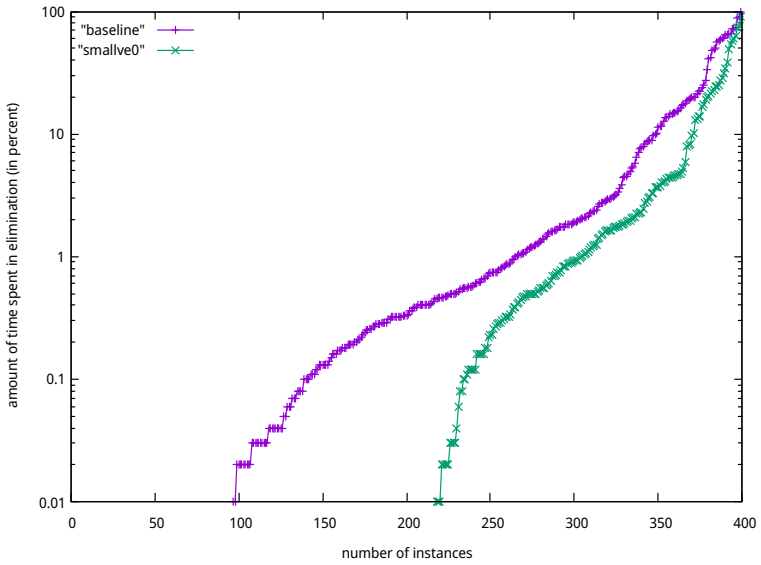


Fig. 2 Percentage of the total amount of time spent in variable elimination in Lingeling

of the co-factors $(F_x|_{\bar{x}})$ and $(F_{\bar{x}}|_x)$ of F, i.e., the formula that is obtained by removing the occurrences of x in F_x and of $\bar{x}$ in $F_{\bar{x}}$ and then conjoining the result. Alternatively one can obtain the candidate formula to be checked for unsatisfiability by removing all occurrences of the literals x and $\bar{x}$ from the environment $E(F, x)$.

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. $\bar{x}$ to the clauses generated in the first step).

Note that we ignore $\Delta(F, x)$ here and focus on environment clauses only. In principle, however, we can replace $\Delta(F, x)$ in F by $(x \vee \Delta(F, x)) \wedge (\bar{x} \vee \Delta(F, x))$ to obtain a CNF (after distributing the variables over $\Delta(F, x)$) where all clauses either contain x or $\bar{x}$. Thus the following discussion extends to the seemingly more general case where also $\Delta(F, x)$ is used as "don't care" for gate extraction.

Let G_ℓ for $\ell \in \{x, \bar{x}\}$ be the identified clauses of F_ℓ mapped back from the clausal core computed by the SAT solver and H_ℓ the remaining clauses, i.e., $F_\ell = G_\ell \wedge H_\ell$. Then it turns out that $F_x \otimes F_{\bar{x}}$ can be reduced to $(G_x \otimes G_{\bar{x}}) \wedge (G_x \otimes H_{\bar{x}}) \wedge (G_{\bar{x}} \otimes H_x)$. In particular $(H_x \otimes H_{\bar{x}})$ can be omitted.[2] The net effect is that fewer resolvents are generated and thus more variables can be eliminated.

To see that non-gate versus non-gate resolvents can be omitted assume that $A \wedge B$ is unsatisfiable and thus $\bar{A} \vee \bar{B}$ is valid. Therefore for any C or D we have

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

With two resolution steps we can then show that the right-hand side implies $(C \vee D)$ and thus can be added to the left-hand side.

$$(A \vee C) \wedge (B \vee D) \;\equiv\; (A \vee C) \wedge (B \vee D) \wedge (C \vee D)$$

Setting $(A, B, C, D) = (G_x|_{\bar{x}}, G_{\bar{x}}|_x, H_{\bar{x}}|_x, H_x|_{\bar{x}})$ shows the rest, more specifically, that $C \vee D = H_{\bar{x}} \vee H_x|_{\bar{x}}$ can be ignored, independent of $A \vee B = G_x|_{\bar{x}}, G_{\bar{x}}|_x$:

$$\begin{aligned}
F_x \otimes F_{\bar{x}} &\equiv (G_x \otimes G_{\bar{x}}) \wedge (G_x \otimes H_{\bar{x}}) \wedge (G_{\bar{x}} \otimes H_x) \wedge (H_x \otimes H_{\bar{x}}) \\
&\equiv (G_x|_{\bar{x}} \vee G_{\bar{x}}|_x) \wedge (G_x|_{\bar{x}} \vee H_{\bar{x}}|_x) \wedge (G_{\bar{x}}|_x \vee H_x|_{\bar{x}}) \wedge (H_x|_{\bar{x}} \vee H_{\bar{x}}|_x) \\
&= (A \vee B) \wedge (A \vee C) \wedge (B \vee D) \wedge (C \vee D) \\
&\equiv (A \vee B) \wedge (A \vee C) \wedge (B \vee D) \\
&= (G_x \otimes G_{\bar{x}}) \wedge (G_x \otimes H_{\bar{x}}) \wedge (G_{\bar{x}} \otimes H_x)
\end{aligned}$$

For the previous example the conjunction of the co-factors of the 7 environment clauses $E(F, x)$ results in the following unsatisfiable formula

$$(\bar{a} \vee \bar{b}) \wedge (a) \wedge (b) \wedge (c) \wedge (d) \wedge (e) \wedge (f).$$

The first three clauses form a clausal core and after adding back x and $\bar{x}$ enable extracting the same gate clauses as before, which in turn enables bounded variable elimination. If only one co-factor contains clauses, e.g., $H_{\bar{x}}$, then we can learn the unit literal x. This rarely happens in our experiments though. This technique is a generalization of failed literal probing [29] where multiple decisions are allowed instead of deciding and propagating just one literal.

[2] Resolvents among gate clauses are not necessarily tautological though (see Section 8).

6 Relating Functional Dependency and Cores

In previous work [28,34,37] the following condition for "definability" was used and we are going to show that in essence it boils down to the same idea. A variable x has a *functional dependency* in F on an (ordered) sub-set of variables D of F with $x \notin D$, i.e., the set D of other variables on which the value of x is functionally dependent, iff the following formula is valid

$$(D = D') \wedge F \wedge F' \;\rightarrow\; x = x' \tag{1}$$

with F' a copy of F where each variable y is replaced by a new variable y'. The intuitive meaning is that there is only one solution for x given the same inputs $(D = D')$, whatever the value of the other variables.

The short-hands $D = D'$ and $x = x'$ denote formulas which enforce that the corresponding original variable and its primed copy assume the same value (through for instance a conjunction of bi-implications). Therefore, there is a functional dependency of x on D iff the following formula is unsatisfiable.

$$(D = D') \wedge F \wedge F' \wedge (\bar{x} = x')$$

The key remark is that $\bar{x} = x'$ and $\overline{x = x'}$ are equivalent because the formula is symmetric in x and x'. In our concrete application, we are not interested in determining the exact set of variables D, because we do not have restrictions on dependencies (unlike in QBF [37] or #SAT [28]). Hence we can pick D, i.e., the variables on which x is supposed to depend, to consist of an arbitrary set of variables occurring in F except x. In practice we will restrict D to the set of variables in the environment of $E(F, x)$ different from x and this way obtain a sufficient but not necessary condition for definability of x over F.

Under this assumption, we prove that our core based condition is the same as definability. First determine CNFs P, N and R such that

$$F \;\equiv\; (x \vee P) \wedge (\bar{x} \vee N) \wedge R$$

where neither x nor $\bar{x}$ occurs in R. Then simplify $(D = D') \wedge F' \wedge (\bar{x} = x')$ to

$$\begin{aligned}
(F \wedge F')[D' \mapsto D][x' \mapsto \bar{x}] &= F \wedge (F'[D' \mapsto D][x' \mapsto \bar{x}]) \\
&= F \wedge \left(((x' \vee P') \wedge (x' \vee N') \wedge R')\,[D' \mapsto D][x' \mapsto \bar{x}]\right) \\
&= F \wedge \left(((x' \vee P) \wedge (x' \vee N) \wedge R)\,[x' \mapsto \bar{x}]\right) \\
&= F \wedge ((\bar{x} \vee P) \wedge (x \vee N) \wedge R)
\end{aligned}$$

using equivalent literal substitution (see for instance [13]). This yields the following satisfiability equivalent formula to our core condition in Eqn. (1)

$$F \wedge (F[x \mapsto \bar{x}]),$$

where on the right x is replaced by its negation $\bar{x}$ and accordingly $\bar{x}$ with x. As F is a CNF this formula contains each clause with x twice, once as in F and once with x (and $\bar{x}$) negated. These two copies of each clause can thus be resolved on x and each resolvent subsumes both antecedents (through self-subsuming resolution). Clauses in F' which do not contain x' nor $\bar{x}'$ become identical after substitution to their counterpart in F.

Therefore the resulting formula after substitution is logically equivalent to the formula obtained from F by removing all the environment clauses $E(F,x)$ (clauses with x or $\bar{x}$) and replacing them with $(F_x|_{\bar{x}}) \wedge (F_{\bar{x}}|_x)$.

To summarize, in order to determine that x is dependent on the variables D in $E(F,x)$ it is sufficient to check unsatisfiability of

$$(F_x|_{\bar{x}}) \wedge (F_{\bar{x}}|_x) \wedge \Delta(F,x)$$

Example 1 (Example of the Proof) Consider the following formula and apply the proof described above: $F = \underbrace{(\bar{a} \vee \bar{b} \vee x)}_{G_x} \wedge \underbrace{(a \vee \bar{x}) \wedge (b \vee \bar{x})}_{G_{\bar{x}}}$ as defined above. The formula

$$\begin{aligned}
&(D = D') && (a = a' \wedge b = b' \wedge c = c') \\
&\wedge F && ((\bar{a} \vee \bar{b} \vee x) \wedge (a \vee \bar{x}) \wedge (b \vee \bar{x}) \wedge (c \vee x)) \\
&\wedge F' && ((\bar{a}' \vee \bar{b}' \vee x') \wedge (a' \vee \bar{x}') \wedge (b' \vee \bar{x}') \wedge (c' \vee x')) \\
&\rightarrow\ x = x'
\end{aligned}$$

is satisfiable iff its negation is unsatisfiable

$$\begin{aligned}
&(D = D') && (a = a' \wedge b = b' \wedge c = c') \\
&\wedge F && ((\bar{a} \vee \bar{b} \vee x) \wedge (a \vee \bar{x}) \wedge (b \vee \bar{x}) \wedge (c \vee x)) \\
&\wedge F' && ((\bar{a}' \vee \bar{b}' \vee x') \wedge (a' \vee \bar{x}') \wedge (b' \vee \bar{x}') \wedge (c' \vee x')) \\
&\wedge \overline{x = x'}
\end{aligned}$$

as the formula is symmetrical in x and x', is unsatisfiable iff the following is too

$$\begin{aligned}
&(D = D') && (a = a' \wedge b = b' \wedge c = c') \\
&\wedge F && ((\bar{a} \vee \bar{b} \vee x) \wedge (a \vee \bar{x}) \wedge (b \vee \bar{x}) \wedge (c \vee x)) \\
&\wedge F' && ((\bar{a}' \vee \bar{b}' \vee x') \wedge (a' \vee \bar{x}') \wedge (b' \vee \bar{x}') \wedge (c' \vee x')) \\
&\wedge \bar{x} = x'
\end{aligned}$$

We replace equivalent variables:

$$\begin{aligned}
&F && ((\bar{a} \vee \bar{b} \vee x) \wedge (a \vee \bar{x}) \wedge (b \vee \bar{x}) \wedge (c \vee x)) \\
&\wedge F'[x' \mapsto \bar{x}] && ((\bar{a} \vee \bar{b} \vee \bar{x}) \wedge (a \vee x) \wedge (b \vee x) \wedge (c \vee \bar{x}))
\end{aligned}$$

Now we resolve each clause of F with its F' counterpart, yielding a clause subsuming its antecedents

$$((\bar{a} \vee \bar{b}) \wedge (a) \wedge (b) \wedge (c))$$

and we can use Kitten to determine that these clauses are unsatisfiable and to produce the following clausal core

$$(\bar{a} \vee \bar{b}) \wedge (a) \wedge (b)$$

In our approach we focus on the environment $E(F,x) \subseteq F$ and only extract definitions implied by $E(F,x)$, which reduces the effort spent in Kitten, but in principle we might want to take additional clauses of F or all of $\Delta(F,x)$ into account to find all definitions (see Example 2 below). We further do not need a conjecture about D a-priori, actually do not even need to determine D for our application at all. It is sufficient to extract gate clauses from the proof of unsatisfiability. Their variables make up D (excluding x).

Example 2 (Missing Environment) Our extraction without additional clauses can miss definitions. Consider for example, the circuit corresponding to $x = a \wedge a = b$, where we add b (resp. $\bar{b}$) to each clause containing x (resp. $\bar{x}$) and are looking for the definition of x. The CNF is $F = (\bar{x} \vee a \vee b) \wedge (x \vee \bar{a} \vee \bar{b}) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b})$. Obviously from F, we know that $x = a$ or $x = b$ are both definitions of x.

$$F_x|_x \wedge F_{\bar{x}}|_{\bar{x}} = (a \vee b) \wedge (\bar{a} \vee \bar{b})$$
$$\Delta(F,x) = (\bar{a} \vee b) \wedge (a \vee \bar{b})$$

Without the additional two clauses in $\Delta(F,x)$, the problem is satisfiable, but becomes unsatisfiable with them. Therefore, our approach without all clauses would miss definability. Remark that in this case, we would actually be able to find the definition of x by first deriving the definition a and eliminating it.

7 Actually Determining the Definition

In order to apply gate information to variable elimination we do not need to extract the actual gate $f(D)$ of x nor need to know the set of input variables D of the gate f. For other applications it might still be interesting to characterize the possibilities of picking f though. Let $L = G|_x$ be the positive co-factor of the gate clauses G and $U = \overline{G|_{\bar{x}}}$ the negation of its negative co-factor, where, to simplify the argument, we use $G_x|_x = G_{\bar{x}}|_{\bar{x}} = \top$, and thus

$$G|_x \;\equiv\; (G_x \wedge G_{\bar{x}})|_x \;\equiv\; G_x|_x \wedge G_{\bar{x}}|_x \;\equiv\; G_{\bar{x}}|_x \;\equiv\; L$$

and

$$G|_{\bar{x}} \;\equiv\; (G_x \wedge G_{\bar{x}})|_{\bar{x}} \;\equiv\; G_x|_{\bar{x}} \wedge G_x|_{\bar{x}} \;\equiv\; G_x|_{\bar{x}} \;\equiv\; \overline{U}.$$

This notation allows us to derive the following "Shannon decomposition" of G:

$$G \;\equiv\; (\bar{x} \vee G|_x) \wedge (x \vee G|_{\bar{x}}) \;\equiv\; (\bar{x} \vee G_{\bar{x}}|_x) \wedge (x \vee G_x|_{\bar{x}}) \;\equiv\; (\bar{x} \vee L) \wedge (x \vee \overline{U})$$

First note that L implies U (written $L \models U$) as $L \wedge \overline{U}$ is the same as $G_{\bar{x}}|_x \wedge G_x|_{\bar{x}}$ and thus unsatisfiable. Now pick an arbitrary f with $L \leq f \leq U$ between the lower bound L and the upper bound U, i.e., $L \models f$ and $f \models U$. We are going to show that $G \models x = f$.

The lower bound gives $\bar{x} \vee L \models \bar{x} \vee f$ and as $G \models \bar{x} \vee L$ we get $G \models \bar{x} \vee f$ by modus ponens. Similarly we have $x \vee \overline{U} \models x \vee \bar{f}$ by contraposition of the upper bound assumption, i.e., $\overline{U} \models \bar{f}$, and derive $G \models x \vee \bar{f}$, which concludes the proof. If f is given explicitly we can pick D as the set of variables occurring in f. If f is given semantically, for instance as function table or BDD, then $y \in D$ iff $f|_y \not\equiv f|_{\bar{y}}$, which can be determined by checking equivalence between co-factors. Similar arguments can be used for characterizing gate extraction from BDDs [20, 41].

8 Resolving Gate Against Gate Clauses

As we have explained above the idea of gate extraction is that we only need to resolve clauses with the definition of the gate. However, we still need to resolve the gate clauses amongst themselves in two cases. First if extracted semantically (Section 8.1). Second if instead of finding a clause, we actually find a *shorter* (subsuming) clause (Section 8.2). Both cases are easy to detect in an implementation.

8.1 Semantical Gate Extraction

Semantic definition extraction does not necessarily produce gate clauses which are tautological, i.e., $G_x \otimes G_{\bar{x}}$ could be non-empty. If these resolvents among gate clauses are not added to the clause set, variable elimination is not satisfiability preserving. Consider the following (unsatisfiable) formula:

$$F \;=\; \underbrace{(x \vee b)}_{G_x} \wedge \underbrace{(\bar{x} \vee a) \wedge (\bar{x} \vee \bar{a} \vee \bar{b})}_{G_{\bar{x}}} \wedge \underbrace{(x \vee \bar{a})}_{H_x} \wedge \overbrace{(\bar{a} \vee c) \wedge (a \vee \bar{b}) \wedge (b \vee \bar{c})}^{\Delta(F,x)}$$

As shown, KITTEN found the (actually minimum unsatisfiable) clausal core $(b) \wedge (a) \wedge (\bar{a} \vee \bar{b})$ in the conjunction of the co-factors of the environment of x, even though there is a shorter core $(a) \wedge (\bar{a})$, which after adding back $\bar{x}$ and x encodes a bi-implication. The reader should be aware that the extracted gate clauses do not encode a NAND gate (second clause has $\bar{x}$ and not x).

This example was produced through fuzzing [15], by comparing a version of KISSAT which correctly resolves gate clauses and one which does not. In this example the fuzzer produced an option setting where extraction of equivalences (bi-implications) was disabled before semantic definition extraction was tried, and then KITTEN simply focused on the larger core.

$$\begin{aligned} G_x \otimes G_{\bar{x}} &= a \vee b \\ G_x \otimes H_{\bar{x}} &= \top \\ G_{\bar{x}} \otimes H_x &= \cancel{(a \vee \bar{a}) \wedge} (\bar{a} \vee \bar{b}) \\ H_x \otimes H_{\bar{x}} &= \top \end{aligned}$$

Thus the correct result after elimination is

$$F'' \;=\; \underbrace{(a \vee b)}_{G_x \otimes G_{\bar{x}}} \wedge \underbrace{(\bar{a} \vee \bar{b})}_{G_{\bar{x}} \otimes H_x} \wedge \overbrace{(\bar{a} \vee c) \wedge (a \vee \bar{b}) \wedge (b \vee \bar{c})}^{\Delta(F,x)}.$$

The last four clauses are satisfiable (setting $a = b = c = \bot$) but the whole F'' as F is unsatisfiable. Therefore the first clause obtained from resolving gate with gate clauses has to be added.

8.2 Syntactical Gate Resolving

We have used fuzzing again to show that the requirement to add gate against gate resolvents is not unique to semantic gate extraction, but also applies to syntactic gate extraction if for instance one allows the solver to use shorter subsuming clauses instead of the exact Tseitin clauses (a common case in Xor extraction [38]). Consider the following encoding of "$x = (\text{if } a \text{ then } b \text{ else } c)$", encoded as:

$$\begin{aligned} G_x &= (x \vee \bar{a} \vee \bar{b}) \wedge (x \vee a \vee \bar{c}) \\ G_{\bar{x}} &= (\bar{x} \vee c) \wedge (\bar{x} \vee \bar{a} \vee b) \\ F' &= (b \vee \bar{a} \vee c) \wedge (a \vee c) \wedge (a \vee \bar{c}) \wedge (\bar{a} \vee \bar{c}) \end{aligned}$$

By resolving on x, we obtain:

$$\begin{aligned} G_x \otimes G_{\bar{x}} &= (\bar{a} \vee \bar{b} \vee c) \wedge (a \vee \bar{b} \vee \bar{c}) \\ G_x \otimes H_{\bar{x}} &= \top \\ G_{\bar{x}} \otimes H_x &= \top \\ H_x \otimes H_{\bar{x}} &= \top \end{aligned}$$

If we do not include the resolvents, then b actually becomes pure and the entire formula is satisfiable with $a = \bot$ and $b = c = \top$. However the formula is actually unsatisfiable. The resolvent of $G_x \otimes G_{\bar{x}}$ contains the clause $\bar{a} \vee \bar{b} \vee c$. By resolving with the first clause $b \vee \bar{a} \vee c$ of F', we obtain the clause $\bar{a} \vee c$ meaning that the clauses are unsatisfiable, because we now have all binary clauses over a and c.

9 Scheduling Variable in the main SAT solver Kissat

Identifying gate clauses syntactically is more efficient than identifying UNSAT cores with a SAT solver, even when using a smaller one like Kitten. Hence, Kissat first uses syntactic pattern matching for a Tseitin encoding of an And, Equivalence, Xor, or IfThenElse gate with the given variable as output, and only if this fails, the inner SAT solver is called. In turn, if this fails due to hitting some limits, the standard elimination criterion is used. This is illustrated in Algorithm 1.

Until 2020, the order of scheduling variables as candidates to be eliminated was done using a priority queue implemented as binary heap, where variables with smaller number of occurrences are tried to be eliminated first. Since the 2021 version, we have (by default) disabled the heap and replaced it with iterating over all active literals; i.e., the variables that have neither been removed nor have already been eliminated. This actually improves performance of Kissat (Figure 3). Of course it avoids updating the heap when removing clauses and probably has other positive effects we still need to investigate in future work.

10 Core-producing lean embedded SAT solver 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 embedded sub-solver Kitten with in-memory proof tracing and fast allocation and deallocation. If

Function `FindGateClauses(`F, x`)`

Input: The clauses F and the variable x
Output: The pair (G, H) of gate and non-gate clauses of F to be resolved

let $E = E(F, x) =$ clauses of F with x or $\bar{x}$
if F contains Tseitin encoding of a gate with output x **then**
  let G be the clauses of the Tseitin encoding of the gate
  return $(G, E \backslash G)$
if call to KITTEN on $(F_x)|_{\bar{x}} \wedge (F_{\bar{x}})|_x$ returns UNSAT **then**
  determine G from clausal core (adding back x and $\bar{x}$)
  return $(G, E \backslash G)$
return $(E, \emptyset)$

Function `BoundedVariableElimination(`F, x, k`)`

Input: The clauses F, the variable x, bound k on additional resolvents
Output: Simplify clauses of F in place if resolvents sufficiently bounded

let $(G, H) =$ `FindGateClauses(`F, x`)`
let $G_\ell =$ clauses of G with $\ell \quad (\ell \in \{x, \bar{x}\})$
let $H_\ell =$ clauses of H with $\ell \quad (\ell \in \{x, \bar{x}\})$
let $R = (G_x \otimes G_{\bar{x}}) \wedge (G_x \otimes H_{\bar{x}}) \wedge (G_{\bar{x}} \otimes H_x)$
let $E =$ clauses of F with x or $\bar{x}$
if $|R| - |E| \leq k$ **then**
  replace F by $R \wedge F'$ where F' are the clauses in F without x nor $\bar{x}$

Algorithm 1: Variable elimination in KISSAT.

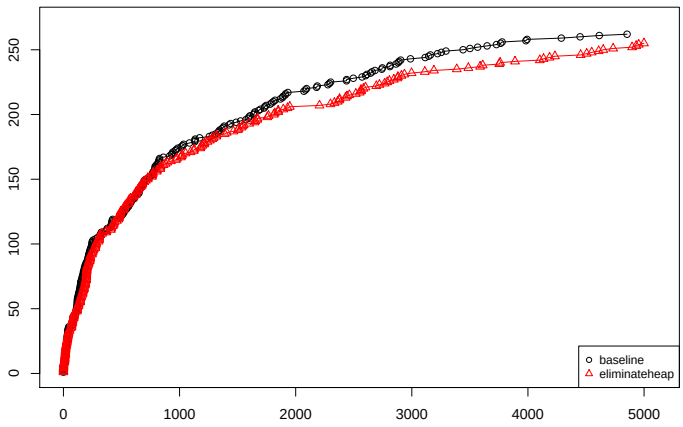


Fig. 3 Performance of KISSAT with and without heap to schedule variable elimination

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, then we can derive a unit clause. 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 [4] discussed at the end of Section 4, which cannot produce DRAT proofs.

Kitten is a very simple SAT solver. Instead of using complicated data structures that take a long time to initialize, Kitten uses watched literals (without blocking literals) and the variable-move-to-front heuristic for decisions. It does not feature garbage collection (no "reduce") nor simplification of added unit clauses. The latter makes it easier to keep track of unsat cores.

To speed up solving and reduce memory usage, Kitten renumbers literals of the given clauses to consecutive literals. Allocations are very fast reusing the internal memory allocator of Kissat instead of allocating new memory. However, even though allocation is fast, it is better to *reuse* the space allocated Kitten within one elimination round. In order to reuse Kitten for the next variable we only clear the necessary content of memory, by for instance clearing stacks for watch lists and the clause arena, instead of deleting and reallocating the solver.

11 Experiments

We have evaluated Kissat on the benchmark instances from the SAT Competition 2020 on 8-core Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (turbo-mode disabled). We used a memory limit of 7 GB (unlike the SAT Competition 2020).

In our first experiment, we have run Kissat with and without gates for variable elimination. The results are presented in Figure 4 and the difference is rather negligible. While the default version performs slightly better, the difference is too small to be significant. However performance is also not worse. The graph also includes the configuration `realloc-kitten-eachtime` where instead of clearing and reusing the same Kitten instance during elimination rounds, Kissat reallocates a new Kitten solver for each variable. Thus avoiding this reallocation turns out to be important at the beginning, even if the impact seems to wear off over time.

We also plotted the amount of time used in the entire elimination procedure (not only the time spent in Kitten). Figure 5 shows that the time spent in Kitten is similar for most problems but in extreme cases is much larger even though the effect is not critical most of the time. However, if we activate preprocessing as described in the next paragraph, we observed extreme cases (like `newpol34-4`) where the elimination took more than 90% of the time. However, these problems are not solved by any Kissat configuration anyhow.

We have further compared efficiency of different techniques by looking at how many variables they have eliminated compared to the total number of eliminated variables (Figure 6). We can see that And-gate elimination is by far the most important, but semantically extracting definitions is second. Extracting IfThenElse gates is not essential. Still, for all extraction techniques, there are a few problems where nearly all eliminated variables are of the given type. We assume that this is due to the structure and the encoding of those problems. Figure 7 shows the same numbers in relation to the total number of variables of the input problem and not compared to the number of eliminated variables, with the same conclusion: And-gate elimination is more important than any other technique.

To evaluate our new elimination technique in more detail, we implemented a preprocessing phase in Kissat, by running explicit preprocessing rounds initially. Each round is composed of probing, vivification, and variable elimination. For

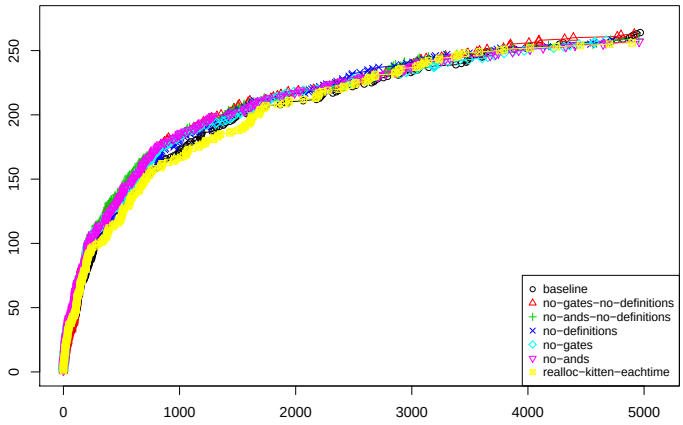


Fig. 4 KISSAT with various options of gates and definitions in variable elimination

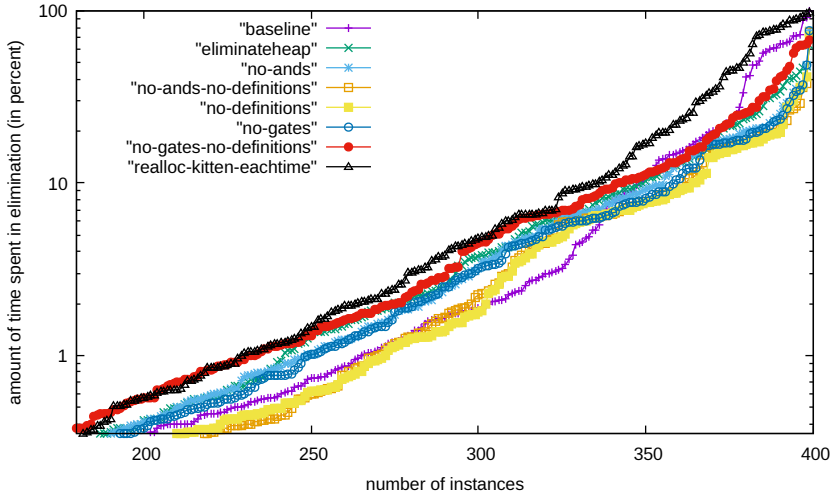


Fig. 5 Percentage of time spent in variable elimination and gate extraction relative to the overall individual running time per benchmark for all 400 SAT Competition 2020 main track instances with time limit 5000 seconds, including benchmarks for which the various versions timed-out. The 100% upper bound on the y-axis reached for some instances means that all time was spent in variable elimination.

our experiments, we use three rounds of preprocessing (or fewer if a fix-point is reached earlier). Then we do not run KISSAT until completion and stop at the first decision. In the default implementation, there is no preprocessing and the same techniques are only called as inprocessing after a few hundred conflicts.

We first compare KISSAT with definitions and gates (the "base line") to the version without definitions. To do so, we show the percentage of removed variables

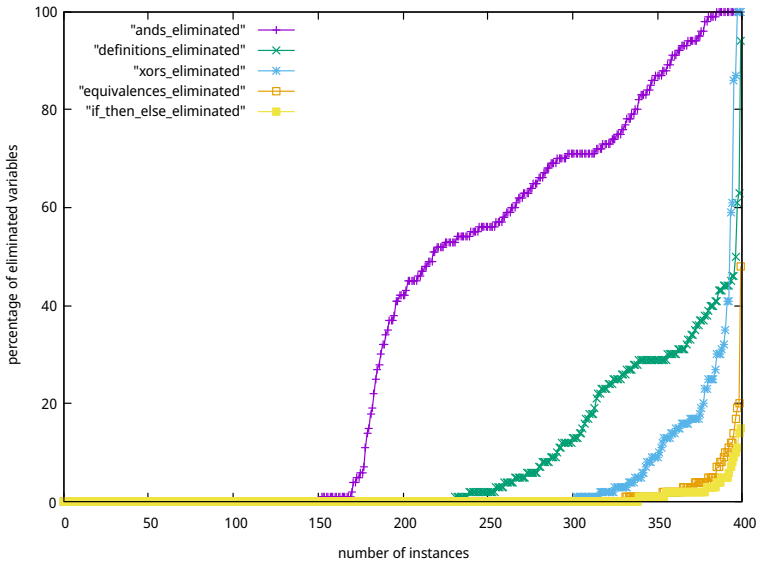


Fig. 6 Percentage of variables eliminated relative to the overall number of eliminated variables of individual benchmarks for all 400 SAT Competition 2020 main track instances, with time limit 5000 seconds, including benchmarks, for which the various versions timed-out. The upper bound 100% on the y-axis reached by some instances means that for all eliminated variables we found (syntactic or semantic) gates and used these during elimination.

in a scatter plot (Figure 8). More variables are eliminated in the version with definitions. In two extreme cases, more than 90% of the variables are eliminated.

An interesting case is deactivating syntactic extraction of gates[3] while keeping definition mining through Kitten (Figure 9). The resulting figure is similar to Figure 8, indicating that Kitten-based definition mining finds those gates too. Note that Kitten does not necessarily find the minimal (smallest) unsat core, nor is it guaranteed to find a minimum core (an MUS). Thus it could in some cases only find large gates even though small gates exists and thus not eliminate as many variables as possible.

The difference in the number of eliminated variables is much higher if we also deactivate `and-gate` detection (Figure 10). With few exceptions the base line removes more variables. Also note that variable elimination is not confluent: eliminating variables in a different order might lead to different results and the number of eliminated variables differs.

Finally, we deactivated syntactic (`no-gates`) as well as semantic (`no-definitions`) gate extraction and compare it to the base line (Figure 11). Much fewer variables are eliminated, as most eliminations need to introduce more clauses.

[3] Using Kissat's `--no-gate` option also deactivates semantic definition extraction. Thus we spelled out all gate types as option in our experiments.

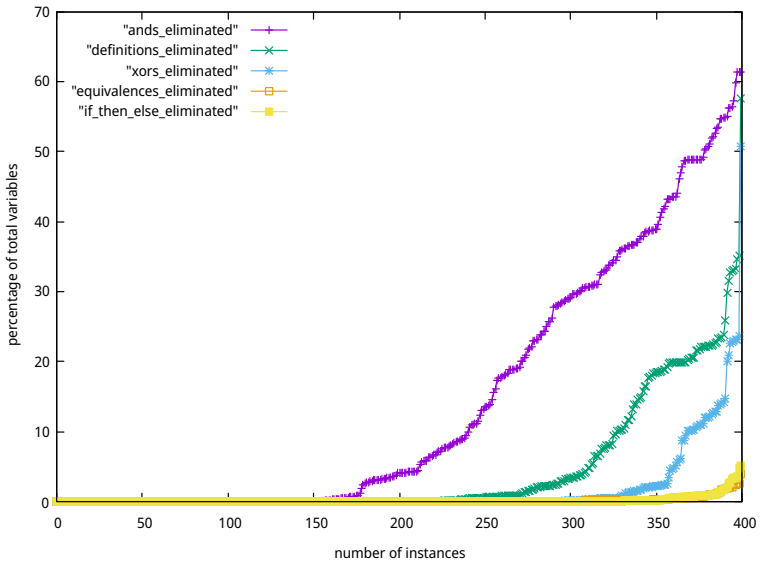


Fig. 7 Percentage of variables eliminated relative to the overall *total* number of variables of individual benchmarks for all 400 SAT Competition 2020 main track instances, with time limit 5000 seconds, including benchmarks, for which the various versions timed-out. The maximum 61% on the y-axis reached by some instances means that 61% of all variables in the input problem have been eliminated by detecting the given gate (syntactic or semantic) during elimination.

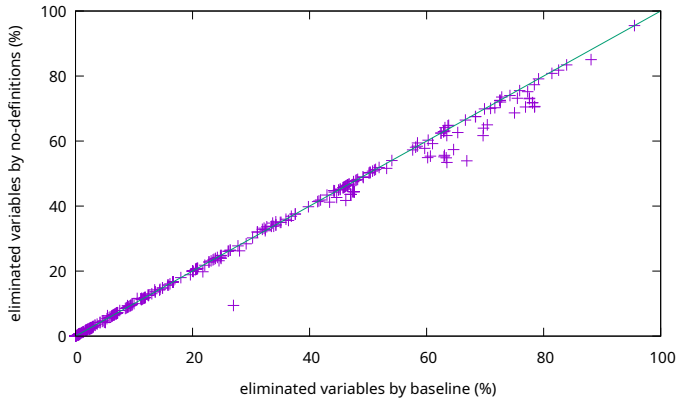


Fig. 8 Deactivating Kitten reduces the number of eliminated variables

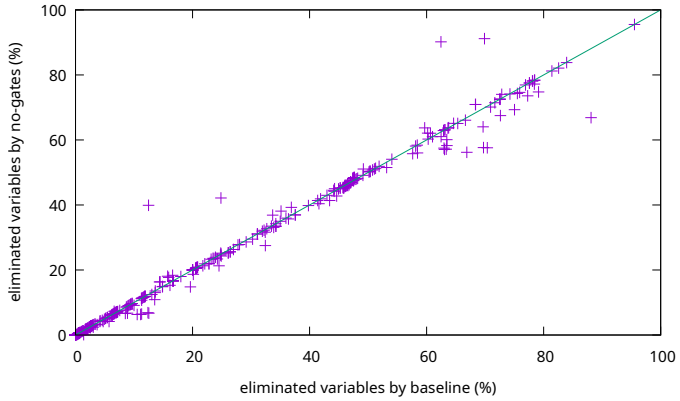


Fig. 9 KISSAT's definition extraction can find gates

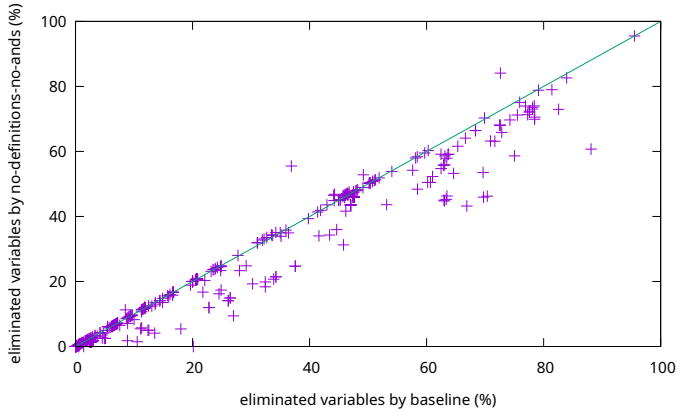


Fig. 10 Deactivating AND-gate detection leads to fewer eliminated variables

12 Related Work

Our approach is mainly motivated by the use of definitions in recent work on model counting [28] and QBF solving [37], where the authors also use core-based techniques, but extract gates explicitly. We showed the connection to this work and claim our restricted formulation is much more concise, because we do not have to extract exactly the variables the definitions depends on.

The approach presented in this article is also the first to use a "little" SAT solver inside a "big" SAT solver to extract definitions, while this related work discussed above uses an ordinary (big) SAT solver to find definitions but for harder problems with a much higher complexity. In circuit synthesis a related approach uses interpolation to find Boolean functions in relations [27].

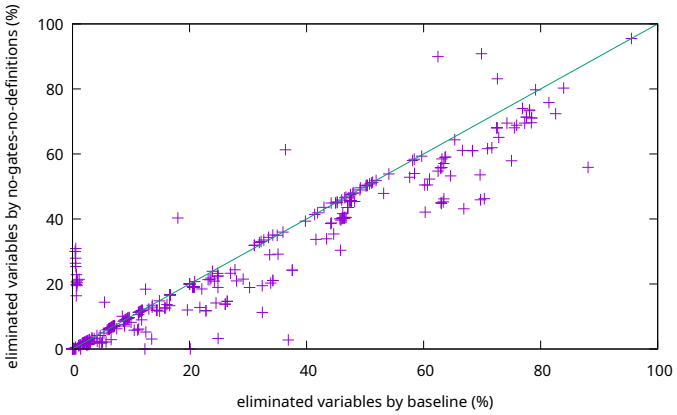


Fig. 11 No gate nor definition extraction compared to KISSAT's base line

Another line of work is related to blocked clause elimination [23, 25], a simplification technique used by SAT solvers to remove clauses. A clause is *blocked* if and only if all resolvents with one literal of the clause are tautologies.

Blocked clauses can be removed from the formula, shifting some work from solving (fewer clauses) to model reconstruction (the model after removal might not be a model anymore). However, detecting gates makes it possible to produce fewer clauses even if the solver subsequently uses BCE. Let's look at the earlier example from Section 4:

$$
\begin{aligned}
F'' \equiv & \\
\cancel{(\bar{a} \vee \bar{b} \vee a) \wedge (\bar{a} \vee \bar{b} \vee b)} \wedge & \quad \text{tautological } G_x \otimes G_{\bar{x}} \text{ resolvents} \\
(\bar{a} \vee \bar{b} \vee e) \wedge (\bar{a} \vee \bar{b} \vee f) \wedge & \quad \text{kept } G_x \otimes H_{\bar{x}} \text{ resolvents} \\
(a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d) \wedge & \quad \text{kept } G_{\bar{x}} \otimes H_x \text{ resolvents} \\
(c \vee e) \wedge (c \vee f) \wedge (d \vee e) \wedge (d \vee f) \wedge & \quad \text{redundant } H_x \otimes H_{\bar{x}} \text{ resolvents} \\
(\bar{c} \vee \bar{d} \vee \bar{e} \vee \bar{f}) & \quad \text{kept } \Delta(F, x)
\end{aligned}
$$

BCE cannot remove the redundant clause $a \vee c$ because it is neither blocked with respect to a (due to clause $\bar{a} \vee \bar{b} \vee e$) nor to c (due to clause $\bar{c} \vee \bar{d} \vee \bar{e} \vee \bar{f}$). By producing fewer clauses during elimination, our method actually makes BCE stronger.

Iser [24] used the "blockedness criterion" to identify gates in addition to a SAT solver (or another approach). He first uses BCE to check that left-uniqueness of the equations, before using the SAT solver to check right-uniqueness. He does not use the SAT solver to identify the clauses, but only to check whether the already identified clauses are right-unique. Iser reports on experiments but does not report on performance changes, only on the amount of time spent in his various strategies.

This work by Iser is also motivated by performing blocked clause decomposition [22], which has the goal to split a CNF in two parts, where the first part is a set of clauses which can be completely eliminated by blocked clause elimination, and the other part contains the remaining clauses. The first "blocked clause set"

is of course satisfiable and models can be generated in linear time. This allows to treat that part almost as a circuit [1]. However, blocked clause decomposition is often costly and the second remaining part of clauses often remains big.

13 Conclusion

We compute cores with a simple little SAT solver Kitten embedded in a large SAT solver Kissat to semantically find definitions after syntactic gate detection fails in order to eliminate more variables. The cost of calling Kitten is limited by focusing on the environment clauses of elimination candidates and its cheap enough to be used whenever syntactic gate detection fails, while it still allows to produce proofs in the DRAT format when needed.

On the considered benchmark set the performance of Kissat is unfortunately not really improved by semantic definition extraction even though the technique is efficient and effective in finding many additional semantic definitions as well as eliminating more variables. The same applies to syntactic gate detection, which in principle is shown to be subsumed by our new semantic approach.

As future work we want to consider further usage of such an embedded SAT solver and started already to apply it to SAT sweeping [12]. We also want to apply our approach and Kitten to extract definitions for preprocessing in model counting and QBF.

Acknowledgment. This work is supported by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE) and the LIT AI Lab funded by the State of Upper Austria. We thank Friedrich Slivovsky for fruitful discussions on Section 6 and Joseph Reeves, Markus Iser, and the anonymous reviewers for comments.

References

1. Balyo, T., Fröhlich, A., Heule, M., Biere, A.: Everything you always wanted to know about blocked sets (but were afraid to ask). In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8561, pp. 317–332. Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_24
2. Biere, A.: About the SAT solvers Limmat, Compsat, Funex and the QBF solver Quantor (2003), presentation for the SAT'03 SAT Solver Competition
3. Biere, A.: Resolve and expand. In: Hoos, H.H., Mitchell, D.G. (eds.) Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3542, pp. 59–70. Springer (2004). https://doi.org/10.1007/11527695_5
4. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Tech. Rep. 10/1, Johannes Kepler University Linz, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2010). https://doi.org/10.350/fmvtr.2010-1
5. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Irwin, M.J. (ed.) Proceedings of the 36th Conference on Design Automation, New Orleans, LA, USA, June 21-25, 1999. pp. 317–320. ACM Press (1999). https://doi.org/10.1145/309847.309942
6. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2

7. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2
8. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
9. Biere, A., Clarke, E.M., Raimi, R., Zhu, Y.: Verifiying safety properties of a Power PC microprocessor using symbolic model checking without BDDs. In: Halbwachs, N., Peled, D.A. (eds.) Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1633, pp. 60–71. Springer (1999). https://doi.org/10.1007/3-540-48683-6_8
10. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Heule, M., Järvisalo, M., Suda, M., Iser, M., Balyo, T. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions (2020), to appear
11. Biere, A., Fleury, M.: Mining definitions in kissat with kittens. In: Workshop on the Pragmatics of SAT 2021 (2021), http://www.pragmaticsofsat.org/2021/
12. Biere, A., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021. In: Heule, M., Järvisalo, M., Suda, M. (eds.) SAT Competition 2021 (2021), submitted
13. Biere, A., Järvisalo, M., Kiesl, B.: Preprocessing in SAT solving. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 391 – 435. IOS Press, 2nd edition edn. (2021)
14. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D.A. (eds.) Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
15. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6175, pp. 44–57. Springer (2010). https://doi.org/10.1007/978-3-642-14186-7_6
16. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods Syst. Des. **19**(1), 7–34 (2001). https://doi.org/10.1023/A:1011276507260
17. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960). https://doi.org/10.1145/321033.321034, http://doi.acm.org/10.1145/321033.321034
18. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3569, pp. 61–75. Springer (2005)
19. Eén, N., Mishchenko, A., Sörensson, N.: Applying logic synthesis for speeding up SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4501, pp. 272–286. Springer (2007). https://doi.org/10.1007/978-3-540-72788-0_26
20. van Eijk, C.A.J., Jess, J.A.G.: Exploiting functional dependencies in finite state machine verification. In: 1996 European Design and Test Conference, ED&TC 1996, Paris, France, March 11-14, 1996. pp. 9–14. IEEE Computer Society (1996). https://doi.org/10.1109/EDTC.1996.494119
21. Fazekas, K., Biere, A., Scholl, C.: Incremental inprocessing in SAT solving. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11628, pp. 136–154. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_9
22. Heule, M., Biere, A.: Blocked clause decomposition. In: McMillan, K.L., Middeldorp, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 19th

International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8312, pp. 423–438. Springer (2013). https://doi.org/10.1007/978-3-642-45221-5_29
23. Heule, M., Järvisalo, M., Lonsing, F., Seidl, M., Biere, A.: Clause elimination for SAT and QSAT. J. Artif. Intell. Res. **53**, 127–168 (2015). https://doi.org/10.1613/jair.4694
24. Iser, M.: Recognition and Exploitation of Gate Structure in SAT Solving. Ph.D. thesis, Karlsruhe Institute of Technology, Germany (2020), https://nbn-resolving.org/urn:nbn:de:101:1-20200429045956607326 48
25. Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6015, pp. 129–144. Springer (2010). https://doi.org/10.1007/978-3-642-12002-2_10
26. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7364, pp. 355–370. Springer (2012). https://doi.org/10.1007/978-3-642-31365-3_28
27. Jiang, J.R., Lin, H., Hung, W.: Interpolating functions from large Boolean relations. In: Roychowdhury, J.S. (ed.) 2009 International Conference on Computer-Aided Design, ICCAD 2009, San Jose, CA, USA, November 2-5, 2009. pp. 779–784. ACM (2009). https://doi.org/10.1145/1687399.1687544
28. Lagniez, J., Lonca, E., Marquis, P.: Definability for model counting. Artif. Intell. **281**, 103229 (2020). https://doi.org/10.1016/j.artint.2019.103229
29. Lynce, I., Silva, J.P.M.: Probing-based preprocessing techniques for propositional satisfiability. In: 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2003), 3-5 November 2003, Sacramento, California, USA. p. 105. IEEE Computer Society (2003). https://doi.org/10.1109/TAI.2003.1250177
30. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, Jr., W.A., Somenzi, F. (eds.) Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2725, pp. 1–13. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
31. Minato, S.: Fast generation of irredundant sum-of-products forms from binary decision diagrams. In: Proceedings of the Synthesis and Simulation Meeting and International Interchange (SASIMI'92). pp. 64–73 (1992)
32. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001. pp. 530–535. ACM (2001). https://doi.org/10.1145/378239.379017
33. Nabeshima, H., Iwanuma, K., Inoue, K.: GlueMiniSat 2.2.10 & 2.2.10-5 (2015), SAT-Race 2015
34. Padoa, A.: Essai d'une théorie algébrique des nombres entiers, précédé d'une introduction logique à une theorie déductive quelconque. In: Bibliothèque du Congrès international de philosophie. vol. 3, pp. 309–365 (1901)
35. Reichl, F., Slivovsky, F., Szeider, S.: Certified DQBF solving by definition extraction. In: Li, C., Manyà, F. (eds.) Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12831, pp. 499–517. Springer (2021). https://doi.org/10.1007/978-3-030-80223-3_34
36. Silva, J.P.M., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: Rutenbar, R.A., Otten, R.H.J.M. (eds.) Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996. pp. 220–227. IEEE Computer Society / ACM (1996). https://doi.org/10.1109/ICCAD.1996.569607
37. Slivovsky, F.: Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12224, pp. 508–528. Springer (2020)
38. Soos, M., Meel, K.S.: BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial

Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. pp. 1592–1599. AAAI Press (2019). https://doi.org/10.1609/aaai.v33i01.33011592
39. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5584, pp. 244–257. Springer (2009). https://doi.org/10.1007/978-3-642-02777-2_24
40. Subbarayan, S., Pradhan, D.K.: NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In: SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings (2004), `http://www.satisfiability.org/SAT04/programme/118.pdf`
41. Yang, B., Simmons, R.G., Bryant, R.E., O'Hallaron, D.R.: Optimizing symbolic model checking for constraint-rich models. In: Halbwachs, N., Peled, D.A. (eds.) Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1633, pp. 328–340. Springer (1999). https://doi.org/10.1007/3-540-48683-6_29

Journal Paper

[1] Shaowei Cai, Xindi Zhang, Fleury, Mathias, and Armin Biere. "Better Decision Heuristics in CDCL through Local Search and Target Phases". In: *J. Artif. Intell. Res.* 74 (2022). I did the implementation in Glucose, the experiments with CaDiCaL, the analysis, and helped in the design in CaDiCaL/Kissat., pp. 1515–1563.

7 Understanding and Improving SAT Solvers

Journal of Artificial Intelligence Research 74 (2022) 1515-1563 Submitted 01/2022; published 08/2022

Better Decision Heuristics in CDCL through Local Search and Target Phases

Shaowei Cai SHAOWEICAI.CS@GMAIL.COM
Xindi Zhang ZHANGXD@IOS.AC.CN
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

Mathias Fleury FLEURY@CS.UNI-FREIBURG.DE
Armin Biere BIERE@CS.UNI-FREIBURG.DE
Albert-Ludwigs-University Freiburg, Freiburg, Germany
Johannes-Kepler-University Linz, Linz, Austria

Abstract

On practical applications, state-of-the-art SAT solvers dominantly use the conflict-driven clause learning (CDCL) paradigm. An alternative for satisfiable instances is local search solvers, which is more successful on random and hard combinatorial instances. Although there have been attempts to combine these methods in one framework, a tight integration which improves the state of the art on a broad set of application instances has been missing. We present a combination of techniques that achieves such an improvement. Our first contribution is to maximize in a local search fashion the assignment trail in CDCL, by sticking to and extending *promising* assignments via a technique called *target phases*. Second, we *relax* the CDCL framework by again extending *promising* branches to complete assignments while ignoring conflicts. These assignments are then used as starting point of local search which tries to find improved assignments with fewer unsatisfied clauses. Third, these improved assignments are imported back to the CDCL loop where they are used to determine the value assigned to decision variables. Finally, the *conflict frequency* of variables in local search can be exploited during variable selection in branching heuristics of CDCL. We implemented these techniques to improve three representative CDCL solvers (Glucose, MapleLcm DistChronoBT, and Kissat). Experiments on benchmarks from the main tracks of the last three SAT Competitions from 2019 to 2021 and an additional benchmark set from spectrum allocation show that the techniques bring significant improvements, particularly and not surprisingly, on satisfiable real-world application instances. We claim that these techniques were essential to the large increase in performance witnessed in the SAT Competition 2020 where Kissat and Relaxed_LcmdCbDl_NewTech were leading the field followed by CryptoMiniSAT-Ccnr, which also incorporated similar ideas.

1. Introduction

The satisfiability problem (SAT) asks to determine whether a given propositional formula is satisfiable or not. Propositional formulas are usually represented in conjunctive normal form (CNF). A growing number of problem domains are successfully tackled by SAT solvers, including electronic design automation (EDA) (Silva & Sakallah, 2000), particularly hardware verification (Prasad, Biere, & Gupta, 2005) and model checking (Vizel, Weissenbacher, & Malik, 2015; Biere & Kröning, 2018), mathematical theorem proving (Heule, Kullmann, & Marek, 2016), AI planning (Kautz & Selman, 1992), and spectrum allocation (Newman, Fréchette, & Leyton-Brown, 2018), among others. Additionally, SAT solvers are also often used as a core component of more complex tools such as solvers for satisfiability modulo theory (SMT) (Barrett, Sebastiani, Seshia, & Tinelli, 2021), which form a crucial component of state-of-the-art program analysis and software verification.

Many approaches have been proposed to solve SAT, but conflict-driven clause learning (CDCL) and local search are the most popular ones. Since their inception in the mid-90s, CDCL-based SAT solvers have been applied, in many cases with remarkable success, to a number of practical applications, because CDCL solvers are so effective in practice.

The local search paradigm is an incomplete method only able to solve satisfiable instances. Local search solvers begin with a complete assignment and iteratively modify it, typically by flipping the value of a single variable, until a model is found or a resource limit (usually time) is reached. Although local search solvers usually have worse performance than CDCL on practical instances, they can be more successful on random and hard combinatorial instances (Li & Li, 2012; Cai, Luo, & Su, 2015; Biere, Fazekas, Fleury, & Heisinger, 2020). Many techniques, including clause learning (Fang & Ruml, 2004) and unit propagation (Hirsch & Kojevnikov, 2005), have been tried to improve local search algorithms but they are still not competitive. Recent studies show that given a promising initial solution for local search helps to improve the performance on some benchmarks (Zhang, Sun, Zhu, Li, Cai, Xiong, & Zhang, 2020; Cai, Luo, Zhang, & Zhang, 2021). In this paper we go one step further and the two components exchange information.

It is usually believed that one limitation of CDCL solvers is that they frequently restart (in order to find short proofs) and therefore they usually work with partial short assignments (Ryvchin & Strichman, 2008; Oh, 2015). We argue that this makes finding complete assignments harder. Comparatively, local search solvers explicitly work on complete assignments.

There have been several attempts to combine both approaches. However, in previous hybrid solvers, both solvers, the CDCL and the local search solver, are opaque to each other, at most exchange some partial information in one direction and therefore usually see each other as a black box. These early hybrid solvers invoke the respective solver according to different situations (Mazure, Sais, & Grégoire, 1998; Habet, Li, Devendeville, & Vasquez, 2002; Letombe & Marques-Silva, 2008; Balint, Henn, & Gableske, 2009; Audemard, Lagniez, Mazure, & Sais, 2010) as discussed in Section 8 on related work.

This work is devoted to a tighter cooperation of CDCL and local search for SAT, with CDCL acting as the main solver and local search mainly used as a tool to improve branching heuristics in the CDCL solver. Occasionally the local search solver finds satisfying assignments too (particularly for random formulas where CDCL performs worse) but, of course,

cannot determine unsatisfiability on its own. In contrast to earlier work, the solvers work hand in hand and information flows frequently in both directions. Our main overall goal is thus to decrease the running time of CDCL solvers on real-world application instances, especially for satisfiable instances, by teaming it up with a local search solver as a "sidekick".

Our first two contributions are inspired by the concept of "promising branches" originating in the Glucose solver, where it was used to schedule, actually prohibit, solver restarts. In Section 3 we explicitly expand such promising branches in two different ways during CDCL solving, instead of just avoiding restarts, as proposed in Glucose. Our vehicle to improve models is to change the heuristic to determine values assigned to selected decision variables. Current state of the art relies on saving the previous value to which a variable was assigned (for instance through propagation) and reuse that value as *saved phase* (Pipatsrisawat & Darwiche, 2007) in case the variable is selected as decision. We present two different mechanisms which in regular intervals try to find improved sets of values.

Our **first contribution** and first mechanism to expand promising branches is called *target phases* and tries to maximize the size of the partial assignment (the trail size) explored during a CDCL run consistent under unit-propagation, forcing the CDCL solver to repeat the phases which led to the previously largest assignment. A larger assignment consistent under unit-propagation is considered an improvement, recorded, and then used for selecting values assigned to decision variables. Targeting these recorded phases forces the CDCL solver to stay close to this assignment with the hope to reach larger and larger assignments, thus gradually increasing the size of the assignment (the trail) until a full consistent thus satisfying assignment is found.

While target phases follow the local search principle to optimize a global criterion locally, our **second contribution** and second mechanism explores these promising branches directly by calling a local search solver on an extension of the current assignment, which is "relaxed", thus complete but not necessarily unit-propagation consistent. Promising branches of sufficient length are extended to a complete assignment by the default CDCL decision heuristic mechanism. We use unit-propagation to complete the model, while ignoring all conflicts along the way, because unit propagations are hard to find for local search. Then, a local search solver is called to find a model nearby. If the local search cannot find a model within a given time limit the CDCL search process resumes.

In order to make use of the effort spent in a failed local search attempt, which did not find a model (the usual case), our **third contribution** consists of saving the best assignment found during local search as new improved set of values and reuse it for the phase selection heuristics during assigning values to decision variables. This use of local search can also be considered as a "rephasing" technique (Section 4), which resets saved phases in regular intervals and thus implements a diversification strategy.

Besides the phases of the best assignment, statistics gathered during local search can provide additional information useful for guiding CDCL. As **fourth contribution** we propose to enhance the CDCL variable selection heuristic by giving more focus to those variables with high activity during local search. The idea is that variables for which it is "difficult" to find a consistent value should be given higher preference to be selected as decision during CDCL, as this might settle their value in a satisfying assignment early on or guide the solver to a short proof of unsatisfiability. As an approximation of this difficulty, we propose to use the variables' *conflict frequency* during local search, that is, its frequency of appearing in

unsatisfied clauses. This information is used to modify the variables' *activity* in the VSIDS heuristic and the variables' *learning rate* in the LRB heuristic (Section 5).

To counteract the effect of losing satisfying assignments during rephasing of parts of the formula, that is, for instance of some disconnected component, we further propose to find autarkies (Section 6) during rephasing. The idea is that a partial assignments which satisfies all clauses it touches allows to remove the touched clauses. We also discuss how to reconstruct models of the original formula from models of the simplified formula.

All these ideas are primarily trying to improve promising partial assignments further and they are indeed, according to our experiments, successful in substantially reducing solving time on satisfiable instances. They can further be implemented and incorporated into a SAT solver in such a way that solving time on unsatisfiable formulas in general degrades only slightly if at all, yielding a clear overall performance gain.

We have implemented our proposed techniques in four state-of-the-art CDCL solvers, including the latest version of Glucose (Audemard & Simon, 2009) (from the SAT Competition 2019), and the winners of the Main track of SAT Competition 2019 and 2020, namely MapleLcmDistChronoBT-DL (Kochemazov, Zaikin, Kondratiev, & Semenov, 2019), and Kissat and CaDiCaL (Biere et al., 2020). The experimental results clearly show that these techniques enable solving a remarkable number of additional instances in the main track benchmarks of the last three SAT Competitions from 2019 to 2021 (following the evaluation guidelines set out by the SAT Practitioner Manifesto (Biere, Järvisalo, Le Berre, Meel, & Mengel, 2020)). Moreover, the improved versions of the three CDCL solvers also give better results on an additional real-world benchmark arising from a spectrum repacking problem in the context of bandwidth auction.

As the experiments clearly show, exploration of promising branches either through target phases or through local search are very helpful to solve satisfiable instances, with a slight degradation on unsatisfiable instances (usually solving 2 or 3 fewer unsatisfiable instances on SAT Competition Benchmarks). Using conflict frequency of variables to enhance the CDCL branching strategy has to a large extent positive effects on satisfiable instances too and gives improvements on few unsatisfiable instances. Overall, our proposed techniques significantly improve the performance of CDCL solvers, leading to a remarkable increase in the total number of solved instances, which we also claim is the main reason for the large jump in performance of the top solvers in the SAT Competition 2020, where Kissat and Relaxed_LcmdCbDl_NewTech were leading the field followed by CryptoMiniSAT-Ccnr, which also incorporated similar ideas. In the latest SAT Competition 2021 variants of Kissat were dominating.

This work combines, on the one hand, our previous (partially unpublished) work on target phases (Biere, 2019), rephasing (Biere, 2017a, 2018), and using local search (Biere, 2019; Soos & Biere, 2019) to improve CDCL assignments which was presented at the workshop on Pragmatics of SAT in 2020 (POS'20) (Biere & Fleury, 2020) and, on the other hand, our paper published at SAT'20 on a deeper integration of local search into CDCL (Cai & Zhang, 2021), including "relaxed" CDCL, local search based rephasing and using conflict frequency to enhance branching heuristics. This publication received a best paper award at SAT'20 and its ideas can be dated back to our ReasonLS solver in the SAT Competitions 2018 (Cai & Zhang, 2018) and four relaxed CDCL solvers in the SAT Competitions

2019 (Cai & Zhang, 2019), and an earlier work on MaxSAT solvers that pass assignments between a decimation algorithm and a local search algorithm (Cai, Luo, & Zhang, 2017).

We discovered that both lines of works, while having been developed independently, have the same underlying ideas and give similar quite remarkable improvements. This is also the reason we decided to join forces on writing this article in order to provide a more complete understanding of the ideas. Compared to our previously reported empirical results we have further implemented both line of works in Glucose and provide more details.

2. Preliminaries

In this section we define the notion of formulas we need (Section 2.1). CDCL is a procedure to solve satisfiability problems in CNF (Section 2.2). In particular we recall how decisions work in most implementations. Unlike CDCL that builds a partial assignment, local search solvers work on adapting full assignments (Section 2.3). Finally, we give the setup for the experiments and the SAT solvers we used for our experiments (Section 2.4)

2.1 Preliminary Definitions and Notations on Formulas

Let $V = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean variables, a *literal* is either a variable x or its negation $\neg x$. A *clause* is a disjunction of literals. A conjunctive normal form (CNF) formula $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ is a conjunction of clauses. For simplicity we assume non-tautological clauses, that is, there is no variable x which occurs positively ($x \in C$) and negative ($\neg x \in C$) in the same clause.

A (partial) mapping $\alpha : V \to \{0, 1\}$ is called an *assignment*. If α maps all variables to a Boolean value, it is *complete*; otherwise, it is a *partial*. The size of an assignment α, denoted as $|\alpha|$, is the number of assigned variables in it. The value of a variable x under an assignment α is denoted as $\alpha[x]$. An assignment α satisfies a clause iff at least one literal evaluates to true under α, and satisfies a CNF formula iff it satisfies all its clauses. A CNF formula F is satisfiable iff there is at least one satisfying assignment. Such a satisfying assignment is also called a *model*. The empty clause $\perp$ is always unsatisfiable, and represents a conflict. SAT is the problem of deciding whether a given CNF formula is satisfiable.

A key procedure in CDCL solvers is *unit propagation*. Whenever a clause has one unset literal and all others false, the unset variable is assigned to satisfy this clause. This process is run until fixpoint or until an *empty clause* (a clause false under the current assignment) is produced, also called *conflict*.

2.2 CDCL Solvers

For the sake of the presentation in this article, see (Marques Silva, Lynce, & Malik, 2021) for a more generic overview on CDCL, we consider CDCL SAT solvers to be composed of a *propagate-and-learn* and a *guessing* part. CDCL solvers do propagate-and-learn eagerly in practice (Section 2.2.1) and implementations do not differ much. However, the guessing policy and also the *search-restart* policy (Section 2.2.2) are both considered to be important for performance and differ across implementations.

Algorithm 1: Typical CDCL algorithm: CDCL(F, α)

```
1  dl ← 0;                    //decision level
2  if UnitPropagation(F, α)==CONFLICT then return UNSAT
3  while ∃ unassigned variables do
       /* PickBranchVar picks a variable to assign not in α          */
4      x ← PickBranchVar(F, α);
       /* PickBranchDirection picks the respective value             */
5      v ← PickBranchDirection(F, x, α);
6      dl ← dl + 1;
7      α ← α ∪ {(x, v)};
8      if UnitPropagation(F, α)=CONFLICT then
9          bl ← ConflictAnalysisAndLearning(F, α);
10         if bl < 0 then
11             return UNSAT;
12         else
13             Backtrack(F, α, bl);
14             dl ← bl;
15 return SAT;
```

2.2.1 Overall Organization

Algorithm 1 shows the standard procedure of a CDCL solver, where α is the current assignment, dl is the current decision level and bl denotes the backtrack level. Arguments to the functions are assumed to be passed by reference.

The *UnitPropagation* procedure performs Boolean constraint propagation on the formula and identifies potential conflicts. Once a conflict is derived, it is analyzed and a clause is derived by the *ConflictAnalysisAndLearning* function. This *learned clause* is then added to the clause database. Finally, *Backtrack* adapts the search to the newly learned clause. The branching heuristics consists of two procedures, where *PickBranchVar* selects a variable to assign and *PickBranchDirection* the respective phase.

Note that Algorithm 1 shows a simplified skeleton of a typical CDCL algorithm. It is still missing several important techniques, including restarts, clause deletion policies, and learned clause simplification, among others, as explained in (Marques Silva et al., 2021).

2.2.2 Decision Heuristics and Backtracking

There is a long history of research on branching heuristics in SAT. The choice of the branching heuristics is still considered today to have a large impact on the performance of SAT solvers. See for instance (Biere & Fröhlich, 2015) for a survey on the effect of branching heuristics. Here we briefly discuss three branching heuristics, that is, variants of function `PickBranchVar` of Algorithm 1, which are relevant to this article.

Variable State Independent Decaying Sum (VSIDS) is known to be the first heuristics to use information from recent conflicts instead of all present clauses (Moskewicz,

Madigan, Zhao, Zhang, & Malik, 2001). It relies on the concept of "variable activity". We describe the version used in MiniSAT (Eén & Sörensson, 2003) and most modern implementations of CDCL. Each variable has an *activity* attached to it. Each time a variable occurs in resolved clauses during conflict analysis, the solver increases its activity. This is referred to as *bumping*, effectively decaying the activity of all other variables. When selecting a branching variable, VSIDS picks the variable with the maximum activity score.

Learning-Rate Branching (LRB) (Liang, Ganesh, Poupart, & Czarnecki, 2016) frames branching as an optimization problem that picks a variable to maximize a metric called *learning rate*. The learning rate of a variable x at interval I is $\frac{P(x,I)}{|I|}$, where I is the sequence of conflicts that occurred between the assignment of x until it transitioned back to unassigned, $P(x, I)$ measures the number of conflicts in I, for which x occurred in at least one clause during resolving the corresponding learned clause, and $|I|$ matches the number of learned clauses generated in interval I. Furthermore, the authors of LRB interpret variable selection as optimization problem which is solved via a multi-armed bandit algorithm.

Move to front (VMTF) (Ryan, 2004; Biere & Fröhlich, 2015) is a heuristic to focus aggressively on literals involved in the most recent conflicts. It provides a simpler and more efficient implementation that focuses on the literals involved in the last conflicts. The idea is to mark the last learned literals as the most important – they are moved to the front of the queue and will be selected next (last in first out).

Phase Saving. Most modern decision heuristics, particularly all presented above, pick a variable first and then use another heuristic to determine its value, or *phase*, that it should be set to (i.e., whether the variable should be decided positively or negatively). This is function `PickBranchDirection` of Algorithm 1. Picking the right phase is actually the most important heuristic for satisfiable instances, because the solver can pick variables in arbitrary order if the phases form already a model (satisfying assignment).

Formerly, some state-of-the-art SAT solvers like Chaff (Moskewicz et al., 2001) used information based on the number of occurrences (Moskewicz et al., 2001) with the aim of increasing the number of satisfied clauses under the current assignment, but this is expensive. Other SAT solvers like MiniSAT (e.g., in the version submitted to the SAT Competition 2005) always set literals to false. Instead of using information on the clauses, phase saving (Pipatsrisawat & Darwiche, 2007) captures information on the search process and caches how variables are set during propagation or backtracking. This saved value is later used to set the phase when deciding that variable, bringing the SAT solver back to a similar region of the search space. This simple (it only requires an array and is cheap to update) and easy-to-calculate heuristic has a quite remarkable positive effect on performance and is now standard in most modern SAT solvers.

With phase saving, the solver focuses on the region of the search space explored before. The heuristic is not only important for satisfiable instances, but also for unsatisfiable instances. For instance, if the formula is composed of independent components, phase saving cheaply allows the solver to focus on one component instead of working on multiple components at the same time. In particular, if the problem includes *disjoint* satisfiable components, the interplay between the decision and phase saving heuristics achieves that each component is solved independently and satisfying assignments of previously solved components are maintained.

Rephasing. Clearly, phase saving should be considered an *intensification* strategy and by applying general heuristic search principles should benefit from complementing it with a corresponding *diversification* strategy. Accordingly, the idea of rephasing is to regularly reset saved phases. In principle, saved phases can be set arbitrarily, as phase selection does not influence correctness nor termination of CDCL.

The SAT solvers PrecoSAT (Biere, 2010) and PicoSAT (Biere, 2010) use a Jeroslow-Wang score (Jeroslow & Wang, 1990) to change the saved phases either on all or on irredundant only) clauses in regular intervals, following a Luby sequence. The motivation is to adapt the saved phases to the current formula. The SAT solver StrangeNight (Soos, 2013) flips values with a certain probability depending on the depth of the assignment. The motivation here is to avoid the heavy-tail phenomenon. Manthey reported experiments (Manthey, 2010, Section 3.1) for the SAT solver Riss (Balint, Belov, Järvisalo, & Sinz, 2015) with negative results. However, to the best of our knowledge, this is the first time that several rephasing heuristics were compared and used.

Restarts. For efficiency of SAT solvers on practical instances, restarts turn out to be important. In particular, fast restarts (Ramos, van der Tak, & Heule, 2011) are now common. Originally "Luby restarts" were heavily used because they are a priori optimal strategy (Luby, Sinclair, & Zuckerman, 1993). However, in recent years, most implementations switched to Glucose-style restarts (Audemard & Simon, 2012a), basically, a requirement for prevailing in the SAT Competition. Biere and Fröhlich's presentation acts as a survey on various restart heuristics (Biere & Fröhlich, 2015). To keep completeness of SAT solvers, either restarts must be delayed more and more (for instance following a Luby sequence), or alternatively the number of clauses kept during database reduction needs to be increased (as it is usually done for Glucose-style restarts).

The potential overhead generated by frequent restarts in performing the same decisions and propagations over and over again can be lessened by cheaply reusing parts of the trail (Ramos et al., 2011). In order to find models, the solver must generate long assignments. For Luby-style restarts, this is realized by means of the (non-monotonically) increasing intervals between successive restarts.

For Glucose-style restarts, the intervals are not necessarily increasing and hence there are no guarantees that long assignments will be generated. To overcome that drawback, as implemented in Glucose (Audemard & Simon, 2012b), restarts can be *blocked* and delayed whenever the current assignment looks *promising*, for example, if the trail length has increased by a predefined factor since the latest restart (Audemard & Simon, 2012b). Delaying restarts is mostly useful for satisfiable problems.

Another option is to alternate restart policies (Oh, 2015) and restart less or even suppress restarts for some time in regular intervals during the search process. The latter was shown to be beneficial for satisfiable instances in particular and can be considered as one corner stone to the large improvement of SAT solvers witnessed in the SAT Competition 2016.

2.3 Local Search Solvers

Local search algorithms (Hoos & Stützle, 2004) explore the search space using a neighborhood relation. They start somewhere in the search space and the space is explored following a neighboring relation until some criterion is met. In the context of SAT, the search space

is the set of complete assignments which is characterized as the set of strings $\{0,1\}^n$, where n is the number of variables in the formula.

For SAT, the most natural neighborhood maps candidate solutions to their set of Hamming neighbors, that is, candidate solutions that differ in exactly one variable, until a model is found. In this view, a step in SAT local search consists of flipping the value assigned to a single variable. A survey by the first author (Cai, 2015) provides more details on local search SAT solvers.

2.4 Experiment Preliminaries

This sections describes the set up to evaluate our proposed methods, including a description of the SAT solvers, benchmarks, running environment and experimental methodology.

Base Solvers. We choose several state-of-the-art CDCL solvers as the base solvers for our studies, namely Glucose v4.0 (Audemard & Simon, 2009), CaDiCaL 0v9 for evaluating target phases, which in essence is the version submitted to the SAT Competition 2020 (Biere et al., 2020), MapleLcmDistChronoBT-DL v2.1 (Kochemazov et al., 2019), and Kissat_sat (Biere et al., 2020). Glucose is a milestone of modern CDCL solvers and has won several gold medals in SAT Competitions. MapleLcmDistChronoBT-DL won the SAT Race 2019 and Kissat_sat won the Main Track of SAT Competition 2020.

We choose CCAnr (Cai et al., 2015) as the local search solver to integrate into the CDCL solvers Glucose and MapleLcmDistChronoBT-DL, while Kissat_sat already includes a simple local search procedure inspired by ProbSAT (Balint & Schöning, 2012) and particularly YalSAT (Biere, 2014). CCAnr is a local search solver with the aim for solving structured SAT instances and has shown competitive results on various structured instances from SAT competitions and applications.

Benchmarks. The experiments are carried out on the main track benchmarks of the SAT Competitions and one SAT Race of the last three years (2019 – 2021). Additionally, we evaluate the solvers on an important application benchmark suite consisting of 10 000 instances[1] from the spectrum repacking in the context of bandwidth auction which resulted in about 7 billion dollar revenue (Newman et al., 2018).

Experiment Setup. We conducted all experiments on a cluster of computers with Intel Xeon Platinum 8153 @2.00 GHz CPUs and 1 024 GB RAM under the operating system CentOS 7.7.1908. For each instance, each solver run with a cutoff time of 5 000 s. For each solver and benchmark year, we report the number of solved SAT/UNSAT instances and the total solved instances, denoted as '#SAT', '#UNSAT', and '#Solved', and the penalized run time average 'Avg' PAR2 score (as used in SAT Competitions), where the run time of a failed run is penalized by twice the cutoff time.

We show the results as tables and a cumulative distribution function (CDF, and not as a cactus plot), that is, as a graph showing the number of solved instances depending on the time. The higher the curve, the better the solver. The source codes[2] and detailed experiment results[3] are available online.

1. https://www.cs.ubc.ca/labs/beta/www-projects/SATFC/cacm_cnfs.tar.gz
2. http://lcs.ios.ac.cn/~caisw/Code/JAIR-SATcodes.zip
3. http://lcs.ios.ac.cn/~caisw/Code/JAIR-SATtables.zip

3. Exploring Promising Branches

CDCL attempts to produce short proofs and hence often restarts. To focus the search towards models, we force the CDCL part to improve models by using target phasing following ideas from local search. This forces CDCL to stay close to the target assignment (Section 3.1). Another way to improve models is to use a local solver directly to explore promising directions (Section 3.2) during the CDCL search.

3.1 Exploring Promising Branches By Directing CDCL

Fast restarts are important for the performance of SAT solvers, but make solving satisfiable instances harder. To mitigate this issue, restarts can be blocked in Glucose. If the search direction is promising (i.e., the current assignment has become much larger), instead of restarting, the search continues (and the conflict count since the last restart is reset, which prohibits restarts for the next 50 conflicts) (Audemard & Simon, 2012b).

The intuition is that promising assignments should be extended towards full assignments (and hopefully a model of the formula) instead of being discarded by restarting. We refine this idea further in our *target phasing* heuristic that saves promising models separately instead of just extending them.

Target Phasing. The target phasing heuristic follows the idea of extending an assignment to a full model. As for phase saving, an additional implicit (but partial) assignment is kept, with the key difference that the target assignment is updated less frequently during the search. It follows an idea from local search: the target is the assignment the solver tries to fulfill and one mutation corresponds to finding a better assignment. Unlike most local search methods, we still use and prioritize unit propagations over the target: propagations ignore the saved target assignment. Only decisions follow the previously saved target phases. More precisely, target phasing consists of the following three parts.

1. First, an implicit target assignment is saved. Whenever the current assignment becomes more promising (better) than the saved one, the latter is replaced. The current assignment, as represented by the "trail" of the solver is more promising if it assigns more variables (in terms of the size of the "trail") without leading to a conflict after propagation. Then the entire current assignment, that is, the trail, becomes the new target assignment. The replacement is done before each decision if there is no conflict.

2. Second, when picking the phase to assign to a decision variable, we do not use the saved value (as usually done with phase saving) but instead the value given by the target assignment. If the target phase of the selected variable is unassigned, the solver defaults to the value provided by phase saving or even to the default phase if the variable was never assigned yet.

3. Third, the target assignment is reset after each rephasing to the initial all-unassigned state. This diversification strategy encourages the solver to find larger and larger target assignments until the next rephasing and has proved to be useful empirically.

Example 1. *To better understand the technique, we give a sketchy example. Assume we start from the empty trail ε, the target phase $\neg B \neg E$, and the saved phase $ABCDEF$. Then*

we decide the variable A. It has no value in the target phases, so we go for the default true *phase. Then we propagate to get the trail* $A^+\neg BC$ *and the saved phases* $A\neg BCDEF$*. We have found no conflict and a more promising model, so the new target phase is* $A\neg BC$*.*

Then we decide another variable D which is set to the default positive value as for A. We get the trail $A^+\neg BCD^+\neg E$ *and the saved phases* $A\neg BCD\neg EF$*. We find a conflict and the trail becomes* $\neg E$*. We now decide another variable B which is set according to its target phase thus to* false *(as* $\neg B$ *was saved as target phase).*

Restart Policy. To increase the chance of finding a model, the solver must work on relatively long assignments. However, this increases the risk of encountering heavy-tailed behavior (Gomes, Selman, Crato, & Kautz, 2000) and to miss short proofs. To circumvent this problem, we alternate between *focused mode* (with Glucose-style fast restarts) and *stable mode* (with fewer restarts)[4] in the spirit of the work by Chanseok Oh (Oh, 2015). Maple is based on Glucose, but the restart mechanism does not include its blocked literals and it has a mechanism to avoid some of the restarts when it is using LRB as decision heuristic.

The 2018 version of CaDiCaL did not restart at all in stable mode. Since 2019, Luby restarts are used with a relatively large base interval (1 024 compared to MiniSAT's default value of 100). The same restart strategy is used for Kissat. Alternating between these two restart policies allows us to use a shorter minimum of conflicts between two successive restarts. Instead of the Glucose default of 50, CaDiCaL has a base restart interval of 2 in focused mode and Kissat even 1 (but increasing logarithmically). The duration of each search mode interval is increased geometrically. In Kissat the conflict interval is in $\mathcal{O}(n \cdot \log^2 n)$ after n mode switches though instead of $\mathcal{O}(n^2)$ (Biere et al., 2020).

3.2 Exploring Promising Branches By Local Search During CDCL

The previous section used CDCL to improve promising models, but a local search solver can achieve the same effect.

First, we provide the motivation of our method. By using reasoning techniques, CDCL solvers are able to prune most of the branches of the search tree. This is useful for solving unsatisfiable instances — to prove a formula is unsatisfiable, a CDCL solver needs to examine the whole search space, and therefore the more of the search tree is pruned, the more efficient the solver is. However, when solving satisfiable formulas, some promising branches are not immediately explored. This makes CDCL solvers miss opportunities for finding a solution. The exploration of promising branches can improve CDCL solvers on satisfiable formulas, and a natural way to do so is to employ local search at such branches.

Now, we present a method to explore promising branches by plugging a local search solver into the CDCL solver, which can improve the ability to find solutions while keeping the completeness of the CDCL solver. The framework of our method is described as follows (Figure 1).

During CDCL, whenever a node is reached with a promising assignment, the search is paused. The algorithm enters a non-backtracking mode, which uses unit propagation and

4. The stable mode was initially called "stable phase" (Biere, 2018), which is a confusing name (due to rephasing), so we have decided to rename it, as can already be seen in the system description of the SAT Competition 2020 (Biere et al., 2020)

a decision heuristic to assign the remaining variables without backtracking, *relaxing* the condition to stop on the first identified conflict clauses. At the end, this leads to a complete assignment β, which the local search solver uses to search for a model nearby. If the local search fails to find a model within a certain time budget, then the algorithm goes back to the normal CDCL search from the node where it was paused (we call this a breakpoint).

We need to identify which branches (i.e., partial assignments) deserve exploration. We propose two conditions below, and any assignment α satisfying at least one of them is considered as promising and will be explored:

- A certain ratio of variables is assigned, that is, $\frac{|\alpha|}{|V|} > p$ and there is no conflict under α, where p is a parameter and is set to 0.4 according to preliminary experiments on a random sample of instances from recent SAT Competitions.
- A length ratio similar to the best saved assignment, that is, $\frac{|\alpha|}{|\alpha\_longest|} > q$ and there is no conflict under α, where q is set to 0.9 for the same reason.

In order to ensure that the search space of adjacent local search calls is sufficiently different, we disallow local search for a certain number of k restarts, where k is set to 500 for Glucose, and 400 for Maple.

As a starting point for local search we first have to create a full assignment. The simplest solution would be to use some saved information. However, it is difficult for local search to achieve unit propagation (as it would require to flip the right literals, making it very unlikely to find propagation chains). Hence we relax CDCL and complete the current partial assignment by alternating decisions and propagations while ignoring all conflicts. Notably, our implementation uses the same Boolean constraint propagation procedure[5] and therefore, also updates the watched literals and the blocking literals of the clause. The current implementation performs unit propagation whenever possible, and decides variables by randomly picking an unassigned variable and assigning a value to it using phase saving (as CDCL does) when propagation cannot continue. Note that the conflicting variables in the relaxed propagation keep their value and are not changed. This approach reuses the propagation loop and phase saving heuristics (although ignores conflicts). We call this approach *relaxed* CDCL because it allows some branches to be extended to a leaf even meeting conflicts, but does not change the data structures and the completeness. We could even reuse the decision heuristic to select variables. Besides the watched literals, the non-backtracking phase does not change the data structures used for CDCL search process.

After obtaining a complete assignment through this relaxed CDCL procedure, the local search solver is called on all problem clauses and all the permanently added learned clauses (i.e., of low LBD and thus heuristically important). On the contrary, Kissat uses only the irredundant ones. However, inprocessing is heavily used, hence it removes subsumed clauses and replaces them by smaller ones. Therefore, the short learnt clauses kept forever in Glucose have a high chance to be in the irredundant clauses, achieving a similar effect.

In general, the time spent in local search has to be limited. To keep the solving process deterministic, we count memory-accesses to estimate time, instead of relying on explicit time limit. Kissat schedules based on the number of memory accesses, but instead of counting each access, the number of cache line accesses is estimated (e.g., accessing a clause

5. Technically we duplicated the code to ignore conflicts, but otherwise there are no differences.

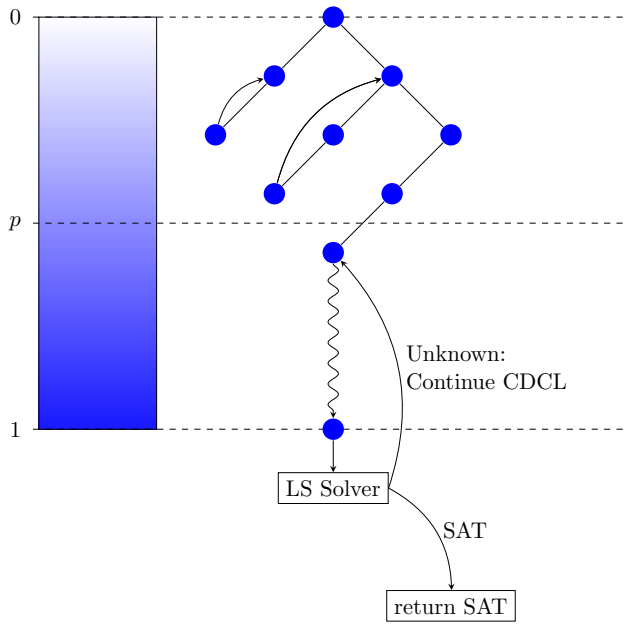


Figure 1: Overall procedure of relaxed CDCL

counts as one, whether one or all elements are evaluated), and the time spent on random walk is increasing in increasing intervals. For our relaxed CDCL implementation, we simply count the accesses to the vector saving the candidate variables. The limit is set to 5×10^7 and fixed during search.

4. Rephasing Heuristics

Exploring promising branches is already a useful addition to CDCL, but it is too stubborn in particular in combination with target phases. In this section, we propose two rephasing heuristics to provide more diversification. By *rephasing* we mean to globally reset or change saved values, and we call a variant of such transformations a *rephasing heuristic*. The first variant uses the improved assignment produced by the local seach solver (Section 4.1). To introduce more scrambling, we also make use of a structured rephasing, for example, setting all phases to true/false (Section. 4.2).

4.1 Local-search Rephasing

In Section 3.2 we proposed a method to plug a local search solver into CDCL solvers, where the CDCL solver helps the local search solver by providing a sensible starting point, from which local search is hoped to find a satisfying assignment in small number of steps. Now,

we propose a rephasing heuristic to import back an improved assignment obtained by the local search process, which is referred to as local-search rephasing (LS rephasing for short).

Algorithm 2: Relaxed CDCL Algorithm with Local-search Rephasing

```
1  dl ← 0, α ← ∅, α_longest ← ∅ ;
2  if UnitPropagation(F, α)=CONFLICT then
3      return UNSAT
4  while ∃ unassigned variables do
5      x ← PickBranchVar(F, α);
6      v ← PickBranchDirection(F, α);
7      dl ← dl + 1;
8      α ← α ∪ {(x, v)};
9      if UnitPropagation(F, α)=CONFLICT then
10         bl ← ConflictAnalysis(F, α);
11         if bl < 0 then
12             return UNSAT
13         else
14             α_longest ← max(α_longest, α);
15             Backtrack(F, α, bl), dl ← bl;
       /* lines 16-22 corresponds to the technique in Section 3.2        */
16     else if (|α|/|V| > p OR |α|/|α_longest| > q) then
17         β ← α;
18         while β is not complete do
19             x ← PickBranchVar(F, β);
20             v ← PickBranchDirection(F, β);
21             β ← β ∪ {(x, v)};
22             UnitPropagation(F, β);
23         if LocalSearch(β, terminate_condition) then
24             return SAT
25     if Meet Restart Conditions then
26         Backtrack(F, α, 0) ;
27         dl ← 0;
28         RephaseFromLocalSearch();          //corresponds to Section 4.1
29 return SAT;
```

Algorithm 2 describes a CDCL solver that implements the idea of exploring promising branches via local search, as well as the LS rephasing heuristic. Every time the CDCL solver restarts (which is forced by a certain schedule and simply backtracks to decision level zero), the LS rephasing heuristic overwrites the saved phases of all variables with assignments produced by local search. To this end, we record the best assignment (with the fewest unsatisfied clauses) in each run of the local search solver, and when we say the assignment of a local search procedure (run), we refer to the best assignment in this procedure.

Phase Name	α_longest_LS	α_latest_LS	α_best_LS	no change
Probability	20%	65%	5%	10%

Table 1: Probability of different phases in our local-search rephasing mechanism.

For our LS rephasing technique, we consider the following assignments, all of which come from the assignments of the local search procedures.

- α_longest_LS: This refers to the assignment of the local search procedure in which the initial solution is extended based on $\alpha\_longest$, where $\alpha\_longest$ is the longest assignment met during past CDCL search. Thus, whenever α_longest is updated, the algorithm calls the local search solver and updates α_longest_LS.
- α_latest_LS: This is the assignment of the latest local search procedure.
- α_best_LS: Among all local search assignments so far, we denote the best one (with the fewest unsatisfied clauses) as α_best_LS.

Local-search Rephasing: Whenever CDCL is restarted, we overwrite the saved phases. We reset all variables with one complete assignment which is selected according to the rephasing probabilities given in Table 1. Such changes are always allowed, because they do not impact the underlying CDCL calculus, its correctness, nor termination.

As can be seen, the LS rephasing considers both intensification and diversification — α_longest_LS and α_best_LS serve for the aim to derive longer models, while α_latest_LS adds diversification, as different local search procedures start with initial assignments built upon different branches. Given how fast restarts are scheduled in modern SAT solvers, the rephasing is done quite often, and with a certain probability (25%), it goes in the directions given by either α_longest_LS or α_best_LS, making it rather aggressive. Overall, it is expected this LS rephasing technique would work well particularly for satisfiable instances, and our experiment results confirm this. To determine the precise percentage we tried every combination (with a 5% increment).

One implementation detail worth mentioning is the restarting policy in Glucose. Its default configuration adapts the strategy according to statistics gathered during the first 100 000 conflicts. We do not change that. However, it blocks restarts (Audemard & Simon, 2012b) as mentioned before. In our first implementation, the restart frequency was so slow that the effect of rephasing was not good. So we remove the blocking restart method from Glucose and only use the LBD quality-based restart policy as Maple to increase the restart frequency and instead rely on our methods to derive longer models. In particular, we removed the restarts from the strategy adaption used by Glucose.

4.2 Fixed Rephasing

The previous section introduced some diversification. However, it is still very search related. In this section, we introduce a more structured version of rephasing that not only considers the past search behavior, but also changes the phases independently to cover more parts of the search space.

4.2.1 Rephasing Options

Rephasing heuristics diversify the exploration of the search space, which can be helpful for satisfiable instances (we could get close to a model). They can also help to variegate learned clauses. We conjecture that this, in turn, improves the efficiency of inprocessing (e.g., learning additional important clauses, like small glue clauses).

Rephasing to a Fixed Value. Our first rephasing heuristic consists in setting all saved phases to a single value, either the original value (phase '`O`') – remember, unlike MiniSat, our tools CaDiCaL and Kissat default to the value *true* – or the opposite value (the inverted value, phase '`I`'). This alternation is helpful in finding models when most values have a certain sign, but this depends on the order in which literals are decided. For example, modifying the SAT solver Glucose (Audemard & Simon, 2009) to apply those rephasings does not make solving certain adversarial factorization problems (Biere, 2017b) completely trivial (some conflicts are still required), but they are now solved extremely fast, while being hard for the default version of Glucose.

Flipping Values. Our second rephasing heuristic consists in flipping the saved values (phase '`F`'), unlike the previous heuristic in which all saved values are set to a single value, namely either *true* or *false*. This allows exploring the "opposite" region of the search space. The motivation behind this heuristic comes from machine learning: Flipping corresponds to diversification with a very different model. It can also support inprocessing by, for example, simulating hyper binary resolution (Heule, Järvisalo, & Biere, 2013), because a literal can be decided and later its opposite.

Randomizing Values. In order to diversify the exploration of the search space even more, we additionally randomize the saved phase (phase '`#`'). The basic idea is that for satisfiable instances and with some luck, the randomized saved phases will now form an assignment that is close enough to a model of the problem we want to solve. The assignment can then be adapted by means of CDCL to a model. If the randomization is done uniformly, the saved phases will eventually be close to a model. In the same spirit, we also tried to shuffle the scores of the variable decision heuristics (and the VMTF queue) in CaDiCaL, which, however, only produced negative results and is switched off by default.

Local Search. With a limited local search (phase '`W`', standing for walk) we attempt to reduce the number of unsatisfied clauses under the assignment formed by the current saved phases or a saved assignment. Our local search implements the ProbSAT strategy (Balint, 2014). During local search an assignment falsifying the least number of clauses is kept as saved phases and this way used for decisions in the CDCL loop. The idea is that the solver can focus on the unsatisfiable part of the clause set. Beside our solvers, CryptoMiniSat (Soos, Nohl, & Castelluccia, 2009) uses local search in a similar way (Soos & Biere, 2019). The search is in essence similar to the one described in the previous section, but it works on a different model: Instead of exploring the current promising assignment while executing CDCL, it alternates between the best model found so far and the current model formed by the saved phases.

This method is very similar to the one described in Section 4.1 but scheduled differently, less frequently, alternating with the other rephasing procedures and originally (and inde-

pendently) employed in the SAT solvers CaDiCaL and Kissat of the last author, Biere, who developed it first.

Best Phasing. The key idea of best phasing (phase '`B`') is that good current assignments are close to models. The solver caches the best assignment found so far (with respect to the length of the trail until the last decision if it generated a conflict). During each backtrack and before each restart, the current partial assignment is saved if it improves the best-so-far found assignment. This heuristic simply replaces the saved phases by the values of the best assignment ever in contrast to target phases which are reset during rephasing. For unsatisfiable instances, this corresponds to focusing on the unsatisfiable region of the search space. This differs from the rephasing heuristic used in the previous section where the model is never reset. The intuition behind that choice is that the best model can get stuck to a local optimum, hence resetting it can help changing the search direction.

Note that the length of the trail might not be a perfect measure in the context of techniques like on-the-fly self-subsuming resolution (OTFS) (Han & Somenzi, 2009; Hamadi, Jabbour, & Sais, 2009) or in combination with chronological backtracking (Nadel & Ryvchin, 2018; Möhle & Biere, 2019). Furthermore, inprocessing needs to be taken into account. In Kissat we actually measure the number of assigned variables plus the number of fixed, substituted, or eliminated variables. The best assignment is reset after each best phasing (in '`B`', and only then).

4.2.2 Rephasing Strategies

The rephasing heuristics define how to change the saved phases, but they do not have to be applied on all variables and an order has to be defined.

Autarkies. As explained above, one motivation for phase saving is that it caches the phases needed to satisfy some components of an input problem allowing the solver to focus on the unsatisfiable part. If saved values are changed by some of the rephasing heuristics described above, this property does not hold anymore potentially harming satisfiable instances that can be split into components. To avoid that, we detect such cases, called *autarkies*, in Kissat, enabling the removal of satisfied components (See Section 6).

Rephasing Strategies. We schedule the different rephasing heuristics in geometrically increasing intervals, unlike the heuristics described in the previous section that are applied after each and every restart. Consequently, we spend more and more time exploring the search space in any given direction. An extreme case would be to start with the inverted phase '`I`' and an infinitely long interval: We would then explore the search space like MiniSAT without any rephasing. We describe the order in which we apply the heuristics in Section 7, but the idea is to apply them in geometrically increasing intervals.

5. Directing the Branching Heuristic with Local Search

In the current presentation, CDCL and the local search solver only exchange assignments, but no information on the search process. In particular, there is no exchange on the variables that are usually involved in conflict clauses, while both solvers use this information: the branching heuristic of CDCL focuses on such variables (the more often it is in a conflict,

the more you should focus on it) and the local search solver prefers to flip those variables. To transmit information to CDCL, we use the conflict frequency of variables in the *latest* local search procedure.

Definition 2 (Conflict frequency)**.** *In a local search process, the conflict frequency of a variable x, denoted as* ls_confl_freq(x)*, is the ratio of flips in which x appears in at least one unsatisfied clause.*

The intuition behind the definition is that the unsatisfied clauses have a similar role to the conflicts during CDCL, so we use that information to adapt the scores in the branching heuristics of CDCL. To update the scores in the branching heuristics, we first multiply ls_confl_freq(x) with a constant integer (100 in this work), and the resulting number is denoted as ls_conflict_num(x). After each restart of the CDCL solver, ls_conflict_num(x) is used to modify the activity score of the variable x for VSIDS and learning rate for LRB.

VSIDS: for each variable x, its activity score is increased by ls_conflict_num(x).

LRB: for each variable x, the number of learned clause during its period I is increased by the number of conflicts ls_conflict_num(x). That is, both $P(x, I)$ and $L(I)$ are increased by ls_conflict_num(x).

The bumping is not done immediately, but only be executed after the next restart, that is, while doing the local-search rephasing described in Section 4.1. We delay rephasing to avoid changing the exploration direction that is done by the CDCL solver currently: As important variables might have been already set, they cannot be decided again, limiting the effect of the search redirecting.

6. Autarky Detection

When the initial problem is composed of several independent subproblems and one such subprolbem is satisfied, the SAT solver will focus on the other parts. With phase saving, the component will remain satisfied. One limitation of rephasing is that the satisfying values for these components are lost. To overcome the issue, it is possible to rephase only some variables without changing the phase of satisfied components or to explicitly identify components, called *autarkies*, that are satisfied by the current assignment and remove them from the overall formula (Section 6.1). If the overall problem is deemed satisfiable, the full model is reconstructed at the end (Section 6.2).

6.1 Algorithm

An autarky is a assignment that fulfills parts of the formula without touching other parts of the formula. This allows for the fulfilled part to be removed from the overall formula without changing the status (SAT or UNSAT). More formally:

Definition 3 (Autarky)**.** *An autarky is an assignment α such that every clause C of F is either entailed ($\alpha \vDash C$) or disjoint ($\alpha \cap \neg C = \emptyset$).*

We use an algorithm originally proposed by Kullmann and described in a publication by Kiesl et al. (Kiesl, Heule, & Biere, 2019). Instead of forcing the autark assignment we simply eliminate the clauses touched by the autarky as well as the variables assigned by it.

Algorithm 3: Algorithm to identify and adapt an autarky from a model

Data: An assignment α and a formula F
Result: An autarky and the adapted formula

```
while there is a clause C ∈ F such that α ⊭ C and α ∩ ¬C ≠ ∅ do
    α := α \ ¬C
foreach literal ℓ ∈ α do
    remove all clauses C from F with ℓ ∈ C;
    add the unit clause ℓ to the reconstruction stack with ℓ as witness;
return (α, F)
```

Algorithm 4: Implementation of the algorithm to identify an autarky from a model

Data: An assignment α and a formula F
Result: An autarky

```
α_S := α;
while α_S ≠ ∅ do
    ℓ := pop α_S;
    foreach clause C in F containing ℓ such that α ⊭ C and ¬C ∩ α ≠ ∅ do
        α_S := α_S ∪ (¬C ∩ α);
        α := α \ ¬C
return α
```

The algorithm is given in Algorithm 3. It is composed of two loops. The first one reduces the current assignment by removing all literals that falsify any clause. If the remaining assignment is not empty, then an autarky was found and the formula can be trimmed by removing all entailed clauses from the set of clauses.

Theorem 4 (Correctness)**.** *Given an assignment α and $\alpha_0 \subseteq \alpha$ be any autarky. After running Algorithm 3, the resulting assignment α' contains α_0 and is an autarky.*

Proof. Let β_i be the updated α after going through the first loop i times. Our first goal is to prove $\alpha_0 \subseteq \beta_i$. Note that $\alpha_0 \subseteq \beta_0$ by assumption, as β_0 is the original α. For the induction step assume $\alpha_0 \subseteq \beta_i$ and that there is still a clause $C \in F$ with $\beta_i \not\models C$ and $\beta_i \cap \neg C \neq \emptyset$. Since $\alpha_0 \subseteq \beta_i$ the first condition shows $\alpha_0 \not\models C$ which implies $\alpha_0 \cap \neg C = \emptyset$ as α_0 is an autarky. Therefore $\alpha_0 = (\alpha_0 \backslash \neg C) \subseteq (\beta_i \backslash \neg C) = \beta_{i+1}$. To prove that α' is an autarky if the loop terminates after n iterations with $\beta_n = \alpha'$ follows immediately as the negation of the loop condition matches the definition of autarky. □

Now, we can prove that Algorithm 3 not only derives an autarky, but the maximal autarky contained in the initial assignment, possibly the empty assignment.

Corollary 5 (Maximal Autarky)**.** *Each assignment α has a unique maximum autarky with $\alpha' \subseteq \alpha$ among all autarkies $\alpha'' \subseteq \alpha$. This maximum α' is computed by Algorithm 3.*

Proof. Assume $\alpha_0, \alpha_1 \subseteq \alpha$ are autarkies. Theorem 4 shows that $\alpha' \supseteq (\alpha_0 \cup \alpha_1)$. Therefore there can not be two different maximal autarkies. □

Algorithm 4 is a refined version with more implementation details of the first loop of Algorithm 3. The solver relies on the occurrence lists to efficiently find all clauses containing a given literal. This is not too costly as redundant learned clauses can be ignored. The algorithm terminates because α_S can contain each literal at most once during execution.

Theorem 6 (Complexity)**.** *The complexity of Algorithm 3 is* $\mathcal{O}(\sum_{C \in F} |C|^2)$.

Proof. The algorithm iterates over every occurrence of a literal in the clauses at most twice during the execution: once if coming from the initial α and potentially a second time if the literal was removed from the assignment α. Therefore, every clause will be read at most twice for each of its literals, and each clause check requires iterating over all the literals. □

The implementation in our SAT solver Kissat is made more efficient by first running the loop of Algorithm 3 *without* adding literals to α_S (called `work` in the actual code[6]). This reduces the number of clauses to visit. Our actual implementation in our SAT solver Kissat is slightly more complicated, because binary clauses are only represented implicitly. Additionally, whenever α becomes empty the execution is stopped immediately as the inner loop condition in Algorithm 4 will be false for all literals.

Performance is further improved by the fact that all irredundant clauses follow each other in memory (without interleaved redundant clauses), improving processor cache efficiency. In our experiments with the SAT solver Kissat, the time spent to identify autarkies is small enough to not be a problem in general and the algorithm can be run until completion, even for very large instances.

6.2 Model Reconstruction

Whenever a non-trivial (non-empty) autarky is found the formula is simplified by removing clauses satisfied by the autarky. Then the solver continues searching for a model. However, if this is successful and a model for the simplified formula is found later, that model does not necessarily satisfy the original formula before applying the autarky assignment and removing the touched clauses.

A similar situation occurs when lifting models to the original formula after variable elimination or removing blocked clauses. The standard solution is to use a *reconstruction stack* on which removed clauses paired with witnesses (in the form of cubes resp. partial assignments) are pushed. During model reconstruction these clauses are consulted and the model is fixed by applying the witness assignment in case such a clause turns out not to be satisfied. This technique was first described in (Järvisalo & Biere, 2010) but goes back to to Niklas Sörensson who proposed it in the context of MiniSAT. For more details please refer to the recent Handbook chapter on preprocessing (Biere, Järvisalo, & Kiesl, 2021).

Thus the implementation of our autarky algorithms also has to properly fill the reconstruction stack. The main question is which witness should be used and whether a single witness clause pair is sufficient. For Algorithm 3 we decided to simply add the literals $\ell \in \alpha$

6. see file `autarky.c` available at `fmv.jku.at/kissat` and in particular the function `propagate_clause` that trims the model for non-binary clauses.

satisfied by the autarky α as units to the reconstruction stack with itself as witness. After removing all the clauses satisfied by the autarky these literals do not occur in the formula anymore and thus we can simply force them to be true during reconstruction.

In principle this has the same effect as learning these units as redundant (PR) clauses as proposed in (Heule, Kiesl, & Biere, 2020) and then removing the same clauses but now because they are satisfied by these units. Unfortunately, this approach would require more sophisticated proof checking, as adding those units is not model preserving, while adding the units only on the reconstruction stack does not influence proof generation and checking.

However, Algorithm 3 is not compatible with incremental SAT solving (Fazekas, Biere, & Scholl, 2019) as those unit witnesses are clearly dependent on each other. It is possible to simply use the full autarky as witness instead, but that blows up the reconstruction stack (quadratically in the worst case) particularly if the autarky is large.

This potentially exploding reconstruction stack is a problem we also saw in the context of globally blocked clauses (Kiesl et al., 2019), also in practice, as well as for covered clause elimination (Barnett, Cerna, & Biere, 2020). We leave it to future work to come up with a space efficient method for autarky reconstruction in incremental SAT Solving. This probably requires a non-clausal reconstruction stack (Barnett & Biere, 2021).

7. Experiments

We have implemented the techniques described in the previous sections in several solvers. Even though technically different, they have the same motivation and share ideas, and all have the same goal to enhance the decision heuristics of CDCL. The results presented in this section show that these improved heuristics yield better performance, particularly for satisfiable instances.

We implemented target phasing and fixed rephasing in CaDiCaL, Glucose, and Kissat (Section 7.1), while the deep combination between CDCL and local search is implemented in Glucose, Maple, and Kissat (Section 7.2). We further combine the techniques of both lines of research in Glucose and Kissat (Section 7.3).

7.1 Directed CDCL

Our heuristics for directed CDCL have been implemented in the SAT solvers CaDiCaL and Kissat and ported to the SAT solver Glucose.

7.1.1 Implementation

Default Policies. We assume that assignments saved either as best or as target phases are good candidates for expansion, and thus finding models faster. Hence, we spend most time on '`B`' phases.

In CaDiCaL the search mode is based on the number of conflicts found so far. In focused mode, the default rephasing policy is '`OI(BWOBWI)`$^\omega$' (`O`riginal, `I`nverted; then `B`est, `W`alk, `O`riginal rephasing is repeated). The '`OI`' at the beginning speeds up finding models where all literals are set to either *true* or *false* (the phase '`I`' starts after the very *first* conflict). In stable mode, the policy is '`(IBWFBW#BWOBW)`$^\omega$' (`F`lipped and `#` for random rephasing). By default, CaDiCaL provides a mode to target satisfiable instances, which only uses target

phasing and stable mode. However, in these experiments, we keep the alternation between stable and focused mode.

In Kissat, the default rephasing policy is '`(BWOBWIBW#BWF)`$^\omega$'. Unlike CaDiCaL, Kissat determines the time spent in focused and stable mode by estimating the number of memory accesses (instead of measuring the time directly in order to be deterministic across runs). More precisely, the number of cache misses is estimated, refining on Knuth's "mems" (Knuth, 2006). Unlike CaDiCaL, in the satisfiable configuration ('`--sat`') submitted to the SAT competition, Kissat alternates between focused and stable mode and always uses target phasing. We have experimented with various parameters to decide how to schedule the rephasing, but the results seems rather robust (even starting with rephasing every 500 conflicts does not lead to worse performance).

All our implementations use the alternation of stable mode with Luby-style restarts and focused mode with Glucose-style restarts. The idea of stable mode is to change less overall. Hence, Kissat and CaDiCaL use chronological backtracking (Nadel & Ryvchin, 2018; Möhle & Biere, 2019). Both solvers also use two separate decision queues as suggested by Oh (Oh, 2015). They use VMTF (Biere & Fröhlich, 2015) in focused and VSIDS in stable mode. VMTF during focused mode makes the solver more agile whereas VSIDS with a low bumping is more stable.

Kissat is the only solver to include autarky detection and elimination. We experimented and found no obvious performance gain or loss. It turns out that autarky detection is fast enough to be executed until completion.

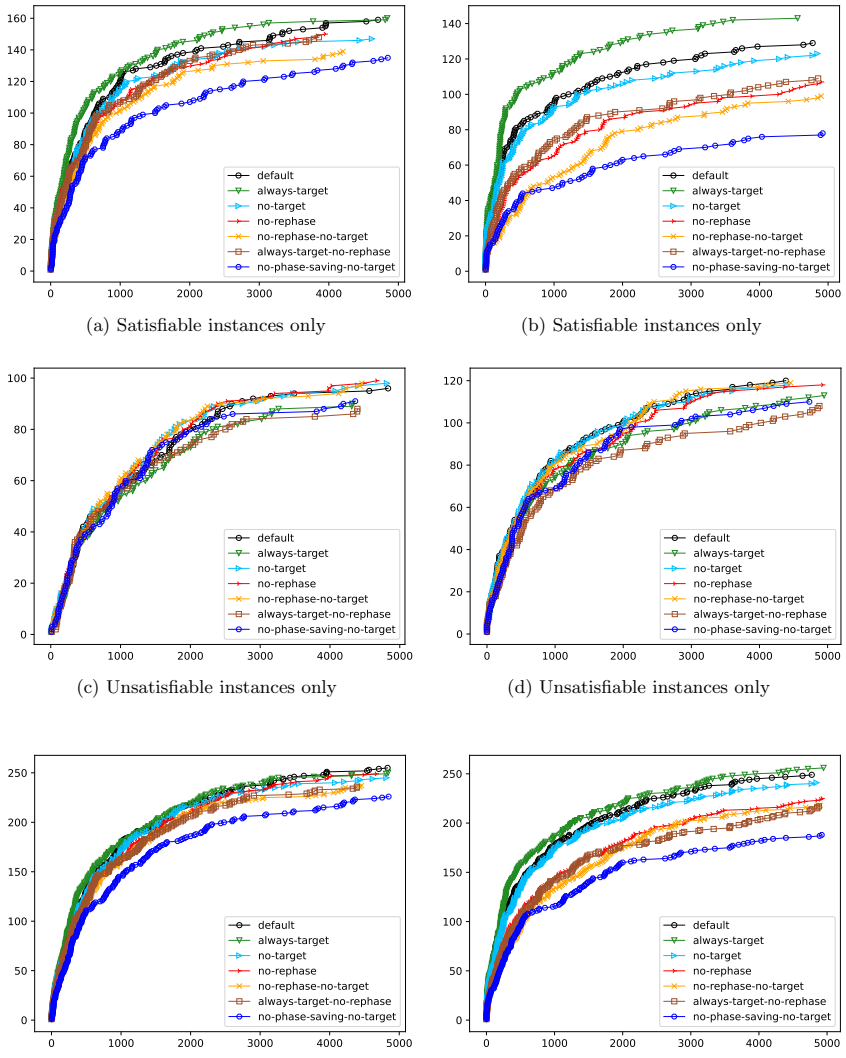
Implementation in Glucose. To have a common platform for comparison between both lines of research, we also implemented our heuristics in the SAT solver Glucose. However, to avoid too many changes to the base solver we did not implement a separate different decision queue for stable mode. Instead, to increase stability, we decrease the bonus that bumped variables get by setting the variable decay to a smaller value. In focused mode we do the opposite and increase the decay compared to the default value in order to follow more closely the search process. To have a more balanced alternation between stable and focused mode, we measure time in the same way as Kissat.

Two details of this implementation effort should be mentioned. First, implementing the alternation of stable and focused mode was easy, but performance significantly dropped to the point that our implementation became much worse than the original implementation of Glucose. We resolved that issue by bumping not only the resolved literals and the literals in the learned clause but also the literals in the *reasons* of the literals in the learned clause, following the idea pioneered by MapleSAT (Liang et al., 2016), which is also used in our other solvers. Experiments with CaDiCaL confirmed the importance of this heuristic. Second, we had to change the types of data structure to save the target phase. Glucose uses a vector of Booleans, but for target phases, it is necessary to use a vector of tri-states (with third possible *unassigned* value, beside *true* and *false*).

7.1.2 Rephasing and Target Phases

For CaDiCaL and Kissat we have tested 7 configurations.

always-target (resp. no-target) always (resp. never) uses target phases to set the value of decision variables. By default, target phasing is only activated in stable mode.



(a) Satisfiable instances only

(b) Satisfiable instances only

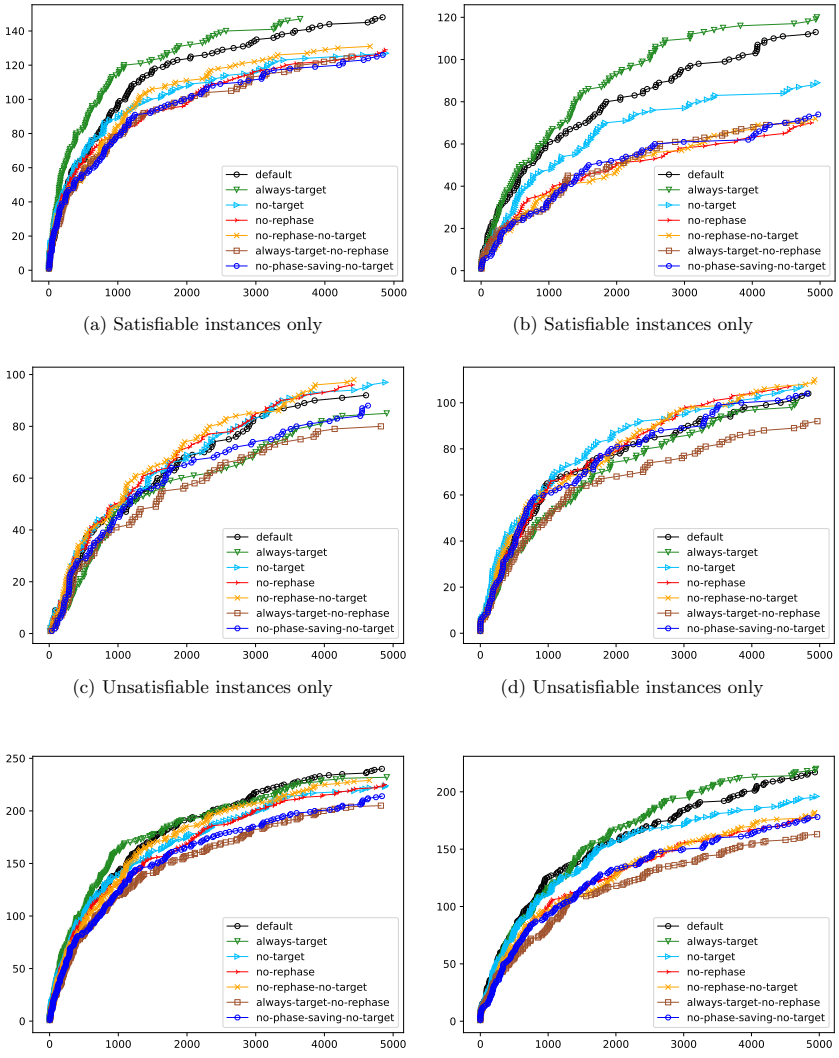
(c) Unsatisfiable instances only

(d) Unsatisfiable instances only

(e) All instances

(f) All instances

Figure 2: CDF for the solver Kissat on benchmarks from the SAT Race 2019 (left) and SAT Competition 2020 (right)



(a) Satisfiable instances only

(b) Satisfiable instances only

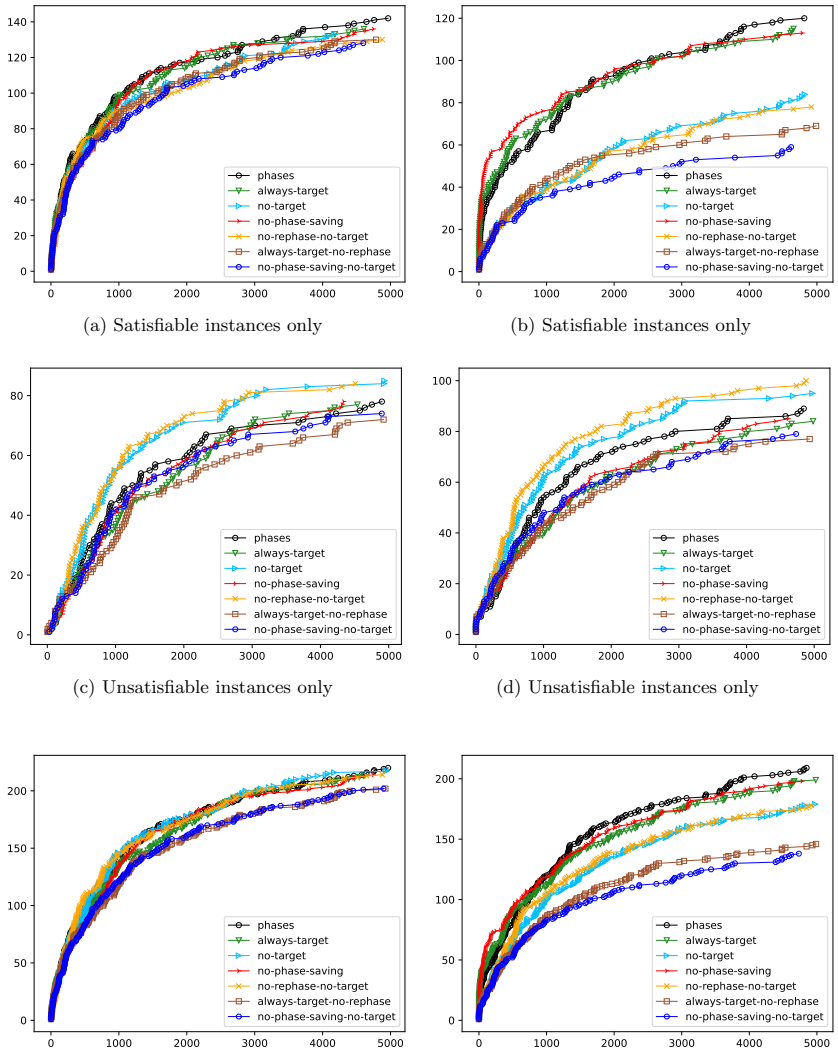
(c) Unsatisfiable instances only

(d) Unsatisfiable instances only

(e) All instances

(f) All instances

Figure 3: CDF for the solver CaDiCaL on benchmarks from the SAT Race 2019 (left) and SAT Competition 2020 (right)



(a) Satisfiable instances only

(b) Satisfiable instances only

(c) Unsatisfiable instances only

(d) Unsatisfiable instances only

(e) All instances

(f) All instances

Figure 4: CDF for the solver Glucose on benchmarks from the SAT Race 2019 (left) and SAT Competition 2020 (right)

no-rephasing never uses rephasing.

default (called **phases**) is an alternating approach. It uses target phasing in stable mode and standard phase saving in focused mode.

no-phase-saving does not use any phase/target saving, nor rephasing, and simply sets the variable to the inital phase, *true*.

For instance, the no-target configuration does not feature any target phasing but uses rephasing, the always-target configuration always uses target phasing even in focused mode, whereas no-rephase uses target phasing only during stable mode but never rephases the save phases. All these configurations use save phasing to save the values.

Results for Kissat and CaDiCaL are presented in Figures 2 and 3 and in Tables 2 and 4. The effect in the SAT Competition 2021 and SAT Competition 2020 are very similar. Thus, for conciseness, we do not include the CDF for the SAT Competition 2021.[7]

First Kissat performs better than CaDiCaL. Second, we see that no-phase-saving solves the least number of problems, confirming previous results. Third, on satisfiable instances, pure phase saving no-target-no-rephase performs worse than any other configuration, but on all instances, it performs better than always-target-no-rephase.

Fourth, target phasing without rephasing (configuration always-target-no-rephase) does not perform better than no target phasing without rephasing no-rephase. Intuitively, this makes sense because target phasing strongly constrains the search in a direction and rephasing resets target phasing making it possible to explore different regions of the search space.

Fifth, our default alternating strategy manages to retain the best side of no-target and always-target: It solves most satisfiable instances and it is not too harmful on unsatisfiable instances. Sixth, no-phase-saving performs particularly bad in Kissat.

Glucose. The results of Glucose give a slightly different picture, see Figure 4 and Table 3, with some interesting results.[8] First, it seems that target phasing degrades performance on unsatisfiable benchmarks, where target phasing is detrimental and our alternating approach is not able to compensate.[9]

Also the impact of the new heuristics is smaller than for Kissat and CaDiCaL. One possible explanation is that further tuning of constants like variable decay during focused and stable mode or a dedicated separated decision heuristic is required to get more out of our heuristics. Third, rephasing is less helpful than for Kissat and CaDiCaL. Fourth, the phases configuration (the same alternating scheduling as for Kissat) solves most SAT problems, albeit by a very small margin. The performance of no-phase-saving is surprising and seems to be due to stable mode: Deactivating it significantly reduces performance on unsatisfiable problems. Remember that *no-phase-saving* uses the alternation between stable and focused mode unlike *original*.

Interpretation. Overall, the performance increases on satisfiable instances. Generally, rephasing with target phasing improves the performance of the solver and makes it more

7. Note to the reviewers, Figures 5 and 6 are part of the appendix.
8. As for the previous case, the CDF from the SAT Competition 2021 is part of the appendix in Figure 7.
9. In our implementation in Glucose 3, this was less detrimental.

solver	#SAT	#UNSAT	#Solved	PAR2
SAT Race 2019(400)				
CaDiCaL-default	148	92	240	4 688.13
CaDiCaL-always-target	147	85	232	4 774.77
CaDiCaL-no-target	127	97	224	5 028.43
CaDiCaL-no-rephase	129	96	225	5 052.92
CaDiCaL-no-rephase-no-target	131	98	229	4 913.18
CaDiCaL-always-target-no-rephase	126	80	206	5 478.14
CaDiCaL-no-phase-saving-no-target	126	88	214	5 313.91
SAT Competition 2020(400)				
CaDiCaL-default	113	104	217	5 316
CaDiCaL-always-target	120	100	220	5 217
CaDiCaL-no-target	90	107	197	5 687
CaDiCaL-no-rephase	70	107	177	6 165
CaDiCaL-no-rephase-no-target	72	110	182	6 094
CaDiCaL-always-target-no-rephase	71	92	163	6 489
CaDiCaL-no-phase-saving-no-target	74	104	178	6 174
SAT Competition 2021(400)				
CaDiCaL-default	115	139	254	4 310
CaDiCaL-always-target	126	134	260	4 243
CaDiCaL-no-target	116	141	257	4 260
CaDiCaL-no-rephase	104	139	243	4 606
CaDiCaL-no-rephase-no-target	107	145	252	4 387
CaDiCaL-always-target-no-rephase	99	122	221	5 151
CaDiCaL-no-phase-saving-no-target	90	137	227	4 966

Table 2: Summary of the performance of the SAT solvers CaDiCaL

robust to solve problems where a model with only *true* variables exists.[10] Performance on unsatisfiable instances degrades slightly except for Glucose.

On the other hand the default alternating approach (target phases during stable mode, usual phase saving during focused mode) achieves a good compromise on a combination of satisfiable and unsatisfiable problems. Rephasing alone does not seem to help as much for Glucose as for the other solvers tested here.

Attempts to unify the rephasing strategies is ongoing work and we did not find a simple overall winning strategy yet. Note that these solvers are of course not identical, (e.g., time spent in stable and focused mode, the number and frequency of deleted learned clauses, the details of the variable scoring mechanism, which inprocessing approaches are used, etc.). Our experience is however, that using best rephasing every second or third time rephasing is scheduled gives better results.

10. This kind of problem is unlikely to be selected at the SAT Competition, because very few solvers are able to solve such instances.

solver	#SAT	#UNSAT	#Solved	PAR2
SAT Race 2019(400)				
Glucose-phases	142	78	220	5 107
Glucose-always-target	136	77	213	5 247
Glucose-no-target	133	85	218	5 116
Glucose-no-phase-saving	136	78	214	5 234
Glucose-no-rephase-no-target	130	84	214	5 184
Glucose-always-target-no-rephase	130	72	202	5 535
Glucose-no-phase-saving-no-rephase	128	74	202	5 525
SAT Competition 2020(400)				
Glucose-phases	120	89	209	5 404
Glucose-always-target	115	84	199	5 624
Glucose-no-target	84	95	179	6 112
Glucose-no-phase-saving	113	85	198	5 585
Glucose-no-rephase-no-target	78	100	178	6 089
Glucose-always-target-no-rephase	69	77	146	6 791
Glucose-no-phase-saving-no-rephase	59	79	138	6 980
SAT Competition 2021(400)				
Glucose-phases	107	117	224	4 963
Glucose-always-target	111	113	224	4 991
Glucose-no-target	103	128	231	4 799
Glucose-no-phase-saving	105	116	221	5 101
Glucose-no-rephase-no-target	106	128	234	4 716
Glucose-always-target-no-rephase	96	99	195	5 655
Glucose-no-phase-saving-no-rephase	83	119	202	5 605

Table 3: Summary of the performance of the SAT solvers Glucose

7.2 Techniques With Local Search

The techniques of deep combination of CDCL and local search include (1) exploring promising branches by local search (denoted as `rx`, Section 3.2); (2) local-search rephasing (denoted as `rp`, Section 4.1) and (3) directing the branching heuristics with local search conflict frequency (denoted as `cf`, Section 5). The experiment setup is described in Section 2.4.

For Glucose and MapleLcmDistChronoBT-DL-v2.1, we implement all the three techniques in this work. For Kissat, we only implement the `cf` technique because it already has a local search solver. We focus on Kissat_sat, the version of Kissat that focuses on satisfiable instances. Nevertheless, it is easy to apply the `cf` technique to Kissat, which is what we do in this work.

Evaluations on Benchmarks of SAT Competitions. The results of evaluations of all the base solvers and the different versions with our techniques are reported in Table 5. The CDFs of these experiments are included in the appendix. According to the results, we have some observations.

solver	#SAT	#UNSAT	#Solved	PAR2
SAT Race 2019(400)				
Kissat-default	159	96	255	4 213
Kissat-always-target	160	90	250	4 255
Kissat-no-target	147	98	245	4 404
Kissat-no-rephase	150	99	249	4 352
Kissat-no-rephase-no-target	139	98	237	4 609
Kissat-always-target-no-rephase	148	88	236	4 607
Kissat-no-phase-saving-no-target	135	91	226	4 939
SAT Competition 2020(400)				
Kissat-default	129	120	249	4 304
Kissat-always-target	143	113	256	4 123
Kissat-no-target	123	118	241	4 497
Kissat-no-rephase	107	118	225	4 974
Kissat-no-rephase-no-target	99	119	218	5 142
Kissat-always-target-no-rephase	109	108	217	5 171
Kissat-no-phase-saving-no-target	78	110	188	5 785
SAT Competition 2021(400)				
Kissat-default	125	151	276	3 638
Kissat-always-target	135	141	276	3 681
Kissat-no-target	120	149	269	3 896
Kissat-no-rephase	114	151	265	3 985
Kissat-no-rephase-no-target	108	149	257	4 164
Kissat-always-target-no-rephase	118	130	248	4 293
Kissat-no-phase-saving-no-target	94	137	231	4 780

Table 4: Summary of the performance of the SAT solvers Kissat

- The `rx` technique improves Glucose and MapleLcmDistChronoBT-DL-v2.1 on solving satisfiable instances, particularly for the benchmarks of 2020 (increased by 17 and 35 for #SAT). On the other hand, the Glucose+rx and Maple-DL+rx have slightly worse performance than the original versions on unsatisfiable instances, and the decrease on #UNSAT is only 2 on average, considering both solvers on all benchmarks.

- By adding the `rp` technique, Glucose+rx+rp and Maple-DL+rx+rp gain further improvement on #SAT, which is significant for all benchmarks. The increase on satisfiable instances is between 6 and 38 problems. Howover, some unsatisfiable instances (less than 4) are lost.

- The impact of the `cf` technique can be seen from the comparisons of Glucose+rx+rp vs. Glucose+rx+rp+cf, Maple-DL+rx+rp vs. Maple-DL+rx+rp+cf, and Kissat_sat vs. Kissat_sat +cf. The results are mixed: On the 2020 benchmarks for Maple the increase is significant for satisfiable instaces. Similar results appear for the 2019 benchmarks with Glucose. Interestingly, the performance for unsatisfiable

solver	#SAT	#UNSAT	#Solved	PAR2
SAT Competition 2019(400)				
Glucose _4.0	115	86	201	5 531
Glucose+rx	120	85	205	5 430
Glucose+rx+rp	131	86	217	5 191
Glucose+rx+rp+cf	143	87	230	4 915
Maple-DL-v2.1	143	97	240	4 602
Maple-DL+rx	146	93	239	4 602
Maple-DL+rx+rp	152	91	243	4 535
Maple-DL+rx+rp+cf	154	95	249	4 377
Kissat_SAT	160	90	250	4 255
Kissat_SAT +cf	163	91	254	4 189
CCAnr1.0	13	0	13	9 678
SAT Competition 2020(400)				
Glucose _4.0	77	93	170	6 325
Glucose+rx	94	90	184	5 939
Glucose+rx+rp	132	91	223	4 942
Glucose+rx+rp+cf	126	98	224	4 978
Maple-DL-v2.1	86	104	190	5 837
Maple-DL+rx	121	105	226	4 978
Maple-DL+rx+rp	141	101	242	4 512
Maple-DL+rx+rp+cf	151	106	257	4 171
Kissat_SAT	143	113	256	4 123
Kissat_SAT+cf	146	113	259	4 055
CCAnr1.0	45	0	45	8 979
SAT Competition 2021(400)				
Glucose _4.0	96	126	222	5 094
Glucose+rx	103	125	228	4 966
Glucose+rx+rp	120	121	241	4 631
Glucose+rx+rp+cf	125	126	251	4 312
Maple-DL-v2.1	104	133	237	4 703
Maple-DL+rx	108	125	233	4 724
Maple-DL+rx+rp	129	121	250	4 364
Maple-DL+rx+rp+cf	130	123	253	4 293
Kissat_SAT	135	141	276	3 681
Kissat_SAT+cf	138	142	280	3 594
CCAnr1.0	24	0	24	9 409

Table 5: Experiment results on benchmarks from SAT Competitions 2019-2021, where Maple-DL-v2.1 is short for MapleLcmDistChronoBT-DL-v2.1

insteances increases back to the original level or is even slightly better. Kissat_sat sees an increase of performance too.

	Glucose-4.2.1		Maple		Kissat_sat		CCAnr
	.	+	.	+	.	+cf	.
#SAT	7 330	8 075	8 084	8 759	8 192	8 214	7 853
#UNSAT	187	197	215	218	207	211	0
#Solved	7 517	8 272	8 299	8 977	8 399	8 425	7 853
Avg (s)	2 555.85	1 850.58	1 867.13	1 243.66	1 760.55	1 734.61	2 215.35

Table 6: Compared with state-of-the-art solvers on FCC. The default version is marked as ".", whereas "+" stands for `+rx+rp+cf`

- By implementing all the three techniques, very large improvements are obtained for Glucose and MapleLcmDistChronoBT-DL-v2.1 for all the benchmarks. Particularly, Glucose+rx+rp+cf solves 54 additional instances than the original solver, and Maple-DL+rx+rp+cf solves 67 additional instances than its original solver for the SAT Competition 2020 benchmark (which has 400 instances). We note that Maple-DL+rx+rp+cf is a simplified and optimized version of our solver Relaxed_LcmdCbDl_NewTech which won the gold medal of Main Track SAT category and the silver medal of the Main Track ALL category in SAT Competition 2020.

Evaluations on Benchmarks of Spectrum Repacking. We also carry out experiments on a suite of instances arising from an important real world project — the spectrum repacking project in US Federal Communication Commission (FCC). The instances of this project was available on-line.[11] (Newman et al., 2018). This benchmark contains 10 000 instances, including both satisfiable and unsatisfiable instances. We compare each base CDCL solver with its final version using our techniques, as well as the underlying local search solver CCAnr.

The results on this benchmark suite are reported in Table 6. According to the results, for each of the base CDCL solvers, the improved version with our techniques has better performance than the base solver. Particularly, the Maple-DL+rx+rp+cf solver solves the most instances (8759+218=8977), significantly better than all the other solvers.

Further Analyses on the Cooperation. We perform more analyses to study the role of local search in the hybrid solvers based on Glucose and MapleLcmDistChronoBT-DL. This experiment does not include Kissat_sat as we do not apply the relaxed CDCL framework to it and the statistics in this experiment are not applicable to Kissat_sat +cf. Some important information is provided in Table 7.

We can see that the local search solver returns a solution for some instances, and this number varies considerably with the benchmarks. A natural question is *whether the improvements come mainly from the complementation of CDCL and local search solvers that they solve different instances?* If this were true, then a simple portfolio that runs both CDCL and local search solvers would work similarly to the hybrid solvers in this work. To answer this question, we compare the instances solved by the hybrid solvers with those by the base CDCL solver and the local search solver (both the CDCL and the local search

11. `https://www.cs.ubc.ca/labs/beta/www-projects/SATFC/cacm_cnfs.tar.gz`

	Analysis for SAT				Analysis for UNSAT	
solver	#byLS	#SAT_bonus	#LS_call	LS_time(%)	#LS_call	LS_time(%)
SAT Competition 2019(400)						
Glucose+rx	9	10	33.94	11.7	22.99	6.95
Glucose+rx+rp	6	19	25.83	10.87	19.78	5.99
Glucose+rx+rp+cf	5	31	26.24	11.75	22.52	6.29
Maple+rx	14	7	12.66	2.67	12.94	1.98
Maple+rx+rp	12	16	12.73	2.91	16.79	2.13
Maple+rx+rp+cf	12	15	11.21	3.05	17.23	2.22
SAT Competition 2020(400)						
Glucose+rx	30	6	15.55	12.2	22.18	11.35
Glucose+rx+rp	21	32	13.67	11.36	12.14	10.57
Glucose+rx+rp+cf	20	31	13.26	11.37	12.65	10.32
Maple+rx	19	13	14.21	6.69	10.24	5.25
Maple+rx+rp	21	30	10.89	6.32	13.09	5.67
Maple+rx+rp+cf	23	36	10.95	6.05	14.17	5.42
SAT Competition 2021(400)						
Glucose+rx	23	7	24.32	13.8	24.9	6.13
Glucose+rx+rp	21	25	16.43	14.07	19.56	5.37
Glucose+rx+rp+cf	17	27	20.1	14.1	14.66	5.53
Maple+rx	17	8	7.47	6.09	5.62	1.69
Maple+rx+rp	17	23	12.84	5.84	6.35	1.71
Maple+rx+rp+cf	14	26	12.73	6.26	5.76	1.69

Table 7: Analyses on the impact of Local Search on the CDCL solvers. Maple is short for Maple-DL to save space, #byLS is the number of instances for which the solution is given by the local search solver, #SAT_bonus is the number of instances for which both base CDCL solver and Local Search solver fail to solve but the hybrid solver finds a satisfiable solution. #LS_call is the average number of calls on Local Search, while LS_time is the average value of the proportion of time (in percentage %) spent on local search in the whole run, and these two figures are calculated for satisfiable and unsatisfiable instances respectively.

solver are given 5000 seconds for each instance). We observe that, there is a large number of instances (denoted by #SAT_bonus) that both CDCL and local search solvers fail to solve but can be solved by the hybrid solvers. For these instances, even a virtual best solver that picks the solver with the best result for each instance would fail. For Glucose, this number reaches 31, 31, and 27 for the three benchmarks respectively, while for MapleLcmDist ChronoBT-DL, this number reaches 15, and 36, and 26 respectively. This clearly indicates that our new cooperation techniques have essential contributions to the good performance of the hybrid solvers.

We have also calculated the number of calls of the local search solver in each run. This figure usually ranges from 10 to 25 calls per run for these benchmarks. As for the run time of local search, which can be seen as the price paid for the benefit of using local search, we calculate the portion of the time spent on local search. This figure is between 6% and 20% for the satisfiable instances, and it drops significantly on unsatisfiable instances, which is usually less than 7%. This is consistent with the observations that the number of local search calls is not necessarily fewer on unsatisfiable instances, because the portion of the time on local search also depends on the total time of the hybrid solver.

On average the time for solving unsatisfiable instances is about 1.5× to 2× the time it takes to solve satisfiable instances for both Glucose+rx+rp+cf and Maple-DL+rx+rp+cf. In a nutshell, the price is acceptable and usually small for the unsatisfiable instances, which also partly explains that our techniques do not have an obvious negative impact on solving unsatisfiable instances although they incline to the satisfiable side.

7.3 Combination of Our Techniques

Finally, we also combined our techniques in a single SAT solver to be able to compare them. Even though they were developed independently, but with the same overall motivation in mind, and also differ on the technical level, the similar positive effects can be observed.

The results are given in Table 8. Overall we can see that both approaches improve the performance of Glucose and in particular on SAT problems. In more details, the (slightly simplified[12]) version of target phasing and rephasing is outperformed by the `+rx+rp+cf`, especially on the SAT 2020 and 2021 benchmarks. Interestingly, the combination of all techniques outperforms both versions.

In an attempt to better understand the phenomenon, we deactivate `+cf` from the combination for Kissat, introducing a performance regression. Understanding the difference better is left to future work.

8. Related Work

There has been interest in combining systemic search and local search for solving SAT. Indeed, it was even included as one of the challenges in Selman et al (Selman, Kautz, & McAllester, 1997). Previous attempts can be categorized into two families according to the type (DPLL/CDCL or local search) of the main body solver.

A family of hybrid solvers use a local search solver as the main body solver. UnitWalk (Hirsch & Kojevnikov, 2005) is among one the first local-search algorithms that try to include a technique mimicing propagation from CDCL, called unit clause elimination. An incomplete hybrid solver hybridGM (Balint et al., 2009) calls CDCL search around local minima with only one unsatisfied clause. Audemard et al. proposed a hybrid solver named Sathys (Audemard, Lagniez, Mazure, & Sais, 2009; Audemard et al., 2010). Each time the local search solver reaches a local minimum, a CDCL solver is launched. Some reasoning techniques or information from CDCL solvers have been used to improve local

12. Missing is in particular the scaling of the random walk at the beginning of walking phases and the better scheduling of which model is extended when walking.

solver	#SAT	#UNSAT	#Solved	PAR2
SAT Competition 2019(400)				
Glucose _4.0	115	86	201	5531.29
Glucose+rx+rp+cf	143	87	230	4915.2
Glucose _phases	142	78	220	5107.06
Glucose+all	151	71	222	5045.35
Kissat	159	96	255	4212.63
Kissat +cf	159	99	258	4157.38
Kissat_SAT	160	90	250	4255.0
Kissat_SAT +cf	163	91	254	4189.04
SAT Competition 2020(400)				
Glucose _4.0	77	93	170	6325.36
Glucose+rx+rp+cf	126	98	224	4977.58
Glucose _phases	120	89	209	5404.23
Glucose+all	142	97	239	4616.08
Kissat	129	120	249	4304.49
Kissat +cf	140	124	264	4042.65
Kissat_SAT	143	113	256	4122.68
Kissat_SAT+cf	146	113	259	4055.31
SAT Competition 2021(400)				
Glucose _4.0	96	126	222	5094.43
Glucose+rx+rp+cf	125	126	251	4312.46
Glucose _phases	107	117	224	4962.6
Glucose+all	130	124	254	4230.77
Kissat	125	151	276	3637.79
Kissat+cf	130	152	282	3562.96
Kissat_SAT	135	141	276	3681.33
Kissat_SAT+cf	138	142	280	3594.41

Table 8: Experiment results of combination techniques on benchmarks from SAT Competitions 2019-2021

search solvers. Resolution techniques were integrated to local search solvers (Cha & Iwama, 1996; Anbulagan, Pham, Slaney, & Sattar, 2005).

Recently, Lorenz and Wörz developed a hybrid solver GapSAT (Lorenz & Wörz, 2020), which used a CDCL solver as a preprocessor before running the local search solver ProbSat. The experiments showed that the learned clauses produced by the CDCL solver were useful to improve the local search solver on random instances.

The other family of hybrid solvers focuses on boosting CDCL solvers by local search, and this work belongs to this line. One simple way of hybridizing is to call local search before CDCL is run, trying to solve the instance by the local search solver alone. This gives the same benefits as a portfolio approach. Additionally, information derived during the local search, such as variable ordering, can be used in the following CDCL solver call. The

hybrid solvers Sparrow2Riss (Balint & Manthey, 2018), CCAnr+Glucose (Cai, Luo, & Su, 2014) and SGSeq (Li & Habet, 2014) belong to this family. In contrast to our approach, there is no information flow back from CDCL to the local search solver. We actually switch between local search and CDCL in regular intervals and further exchange information in both directions in an "inprocessing" fashion (Järvisalo, Heule, & Biere, 2012).

Some works use local search to find a subformula for CDCL to solve. The local search solver (Mazure et al., 1998) finds a part of the formula which is satisfiable, which helps to divide the formula into two parts for the DPLL solver to allow the SAT solver to focus on the unsatisfiable part. In Hinotos (Letombe & Marques-Silva, 2008), a local search identifies a subset of clauses to be passed to a CDCL solver in an incremental way.

Although these previous attempts have been made to combine the strength of CDCL and local search, they did not lead to hybrid solvers essentially better than CDCL solvers on application instances. This work, for the first time, meets the standard of the challenge "create a new algorithm that outperforms the best previous examples of both approaches" (Selman et al., 1997) on standard application benchmarks from SAT Competitions.

9. Conclusion

This work takes a large step towards deep cooperation of CDCL and local search by presenting four techniques for effectively using local search to improve CDCL solvers. The first idea extends promising branches from being pruned by targeting phases of large consistent assignments. The second idea relaxes CDCL by extending such promising branches in order to let local search find a satisfying assignment nearby. The third idea is to utilize assignments minimizing the number of unsatisfiable clauses found during local search and use them as saved phases in the phase selection heuristic. Finally, we proposed to enhance the branching strategy of CDCL solvers by considering the conflict frequency of variables in the local search process. These techniques significantly improve the performance of state-of-the-art CDCL solvers on real-world application benchmarks. As generic techniques they are expected to improve other CDCL solvers too.

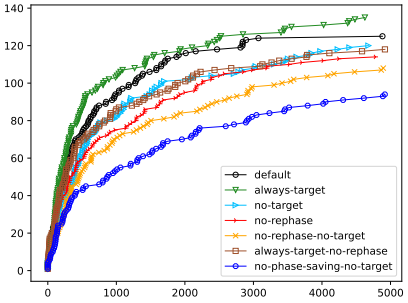
This is the first time that the combination of stochastic search and systematic search techniques leads to substantial improvement of the state of the art on application benchmarks, compared to using only one technique alone, thus positively resolving Challenge 7 of the "*Ten Challenges in Propositional Reasoning and Search*" (Selman et al., 1997).

Acknowledgement

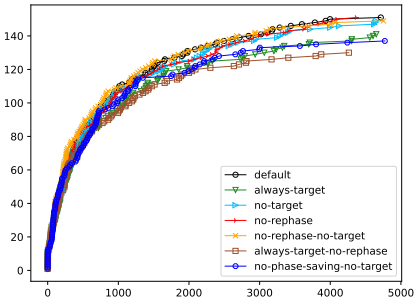
This work is supported by NSFC Grant 62122078, Beijing Academy of Artificial Intelligence (BAAI), the Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), and the LIT AI Lab funded by the State of Upper Austria. Sibylle Möhle and the anonymous reviewers suggested many textual improvements on previous versions of this work.

Appendix

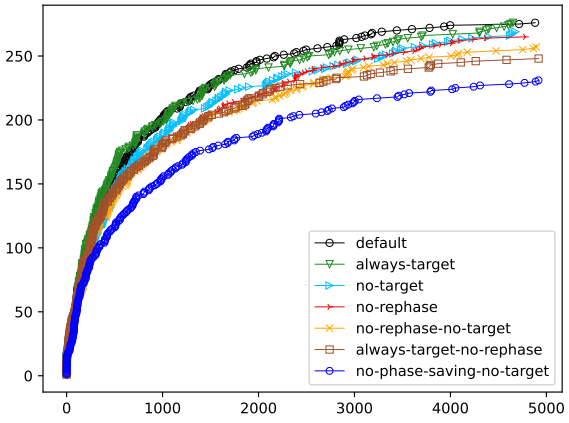
In this section, two classes of CDF plots are listed for component effectiveness analysis. The first three figures evaluate the directly CDCL guided exploring methods. Figures 5-7 show the results of different strategy combinations on the top of Kissat, CaDiCaL and Glucose respectively. The last three figures compare the effectiveness of the local search related strategies, which are implemented based on Glucose and Maple. Figures 8-10 show the results of the different benchmarks form SAT RACE 2019, SAT Competition 2020 and SAT Competition 2021.



(a) Satisfiable instances only



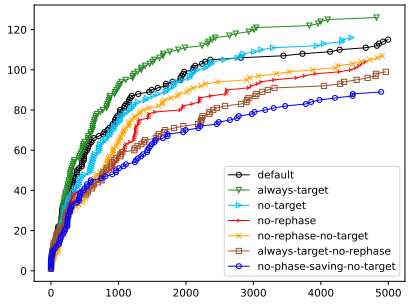
(b) Unsatisfiable instances only



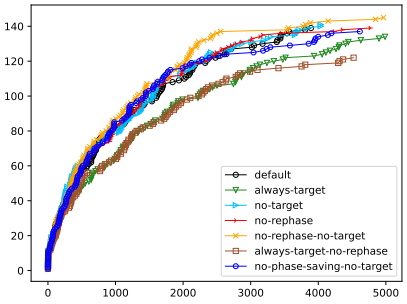
(c) All instances

Figure 5: CDF for the solver Kissat on benchmarks from the SAT Competition 2021

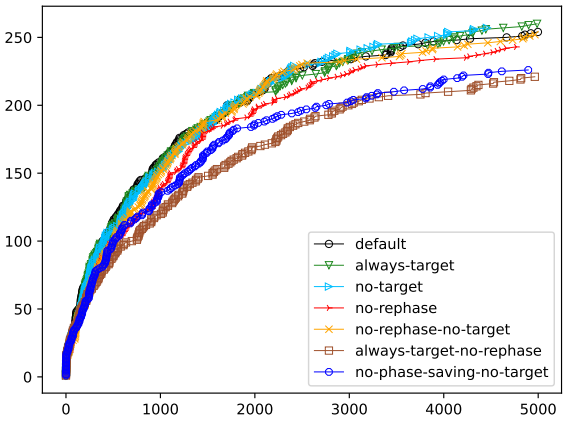
Better Decision Heuristics in CDCL through Local Search and Target Phases



(a) Satisfiable instances only

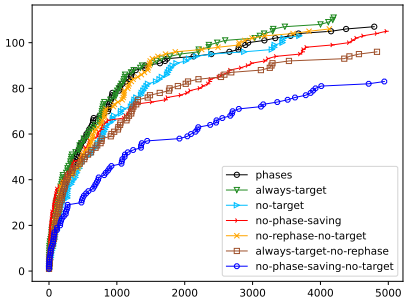


(b) Unsatisfiable instances only

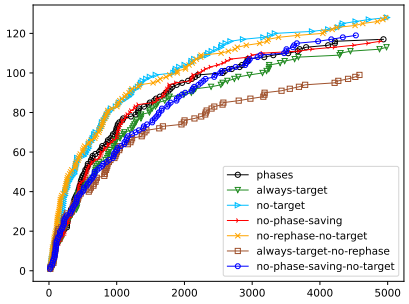


(c) All instances

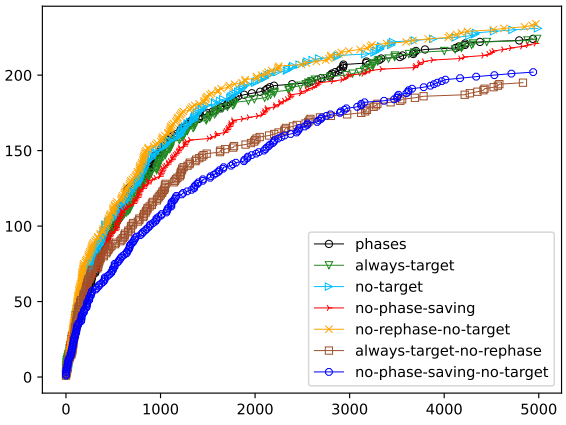
Figure 6: CDF for the solver CaDiCaL on benchmarks from the SAT Competition 2021



(a) Satisfiable instances only

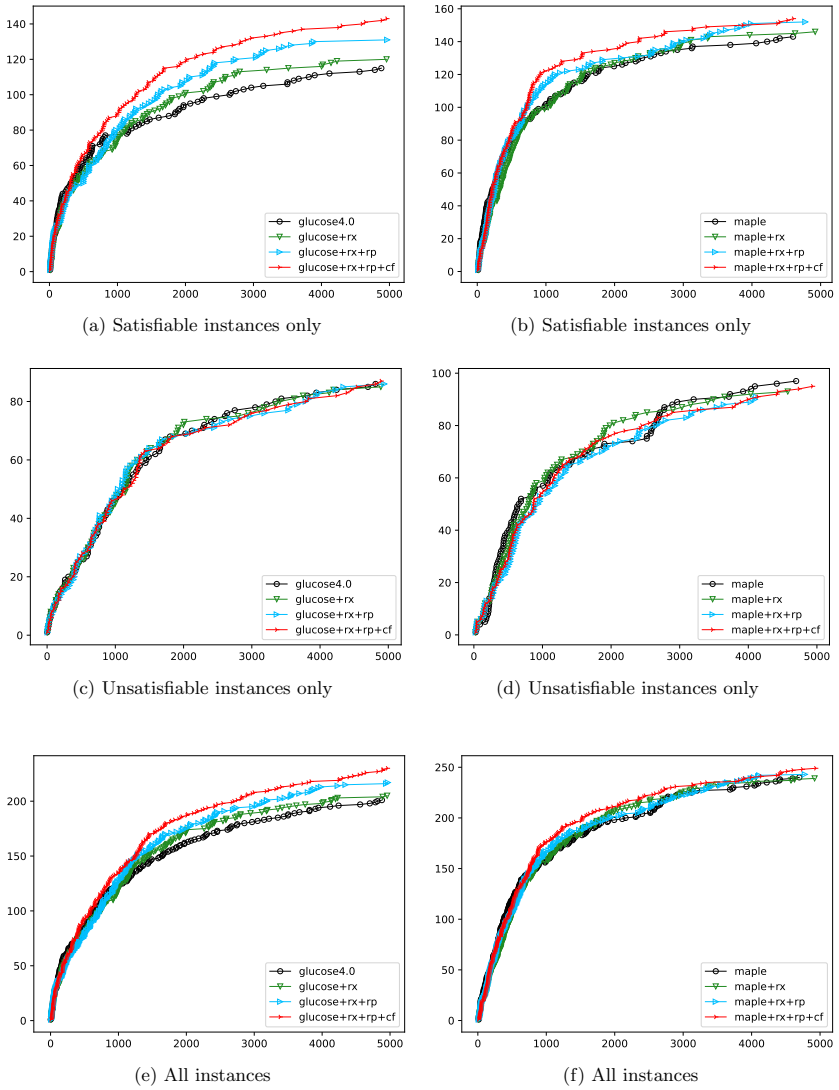


(b) Unsatisfiable instances only



(c) All instances

Figure 7: CDF for the solver Glucose on benchmarks from the SAT Competition 2021



(a) Satisfiable instances only

(b) Satisfiable instances only

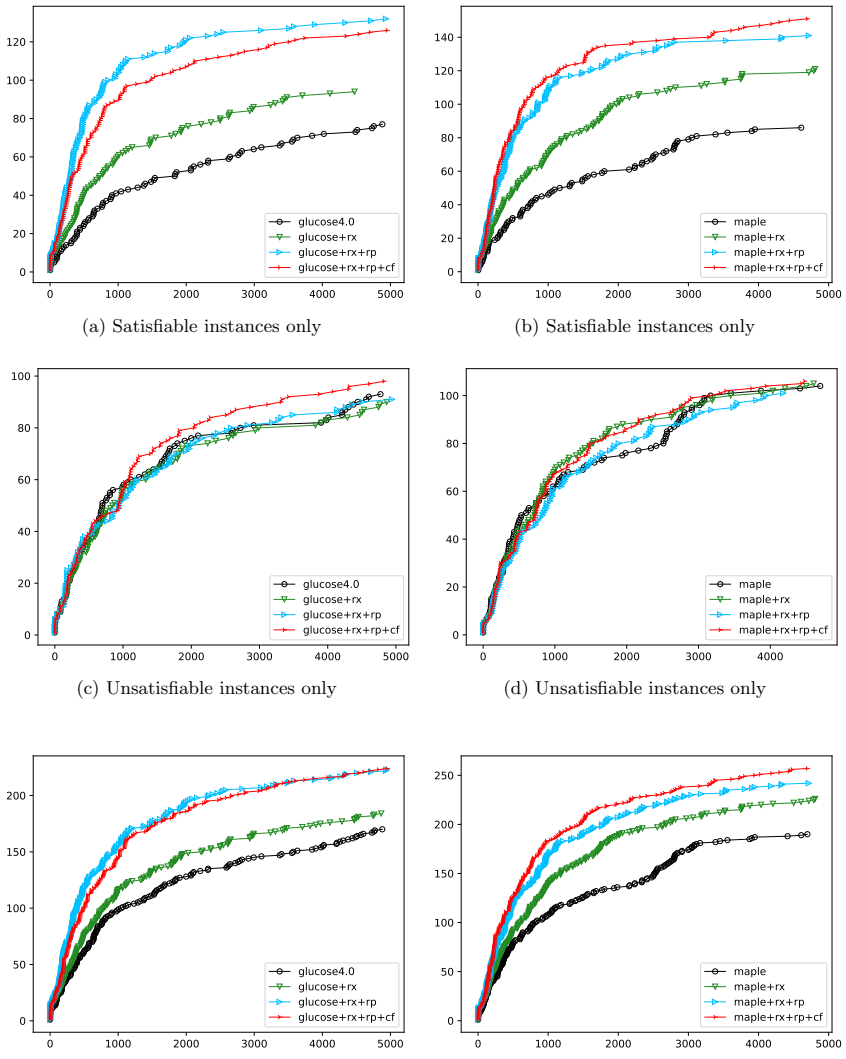
(c) Unsatisfiable instances only

(d) Unsatisfiable instances only

(e) All instances

(f) All instances

Figure 8: CDF for the solvers Glucose (left) and Maple (right) about the relaxed CDCL, local search rephasing and conflict frequency on benchmarks from the SAT Race 2019



(a) Satisfiable instances only

(b) Satisfiable instances only

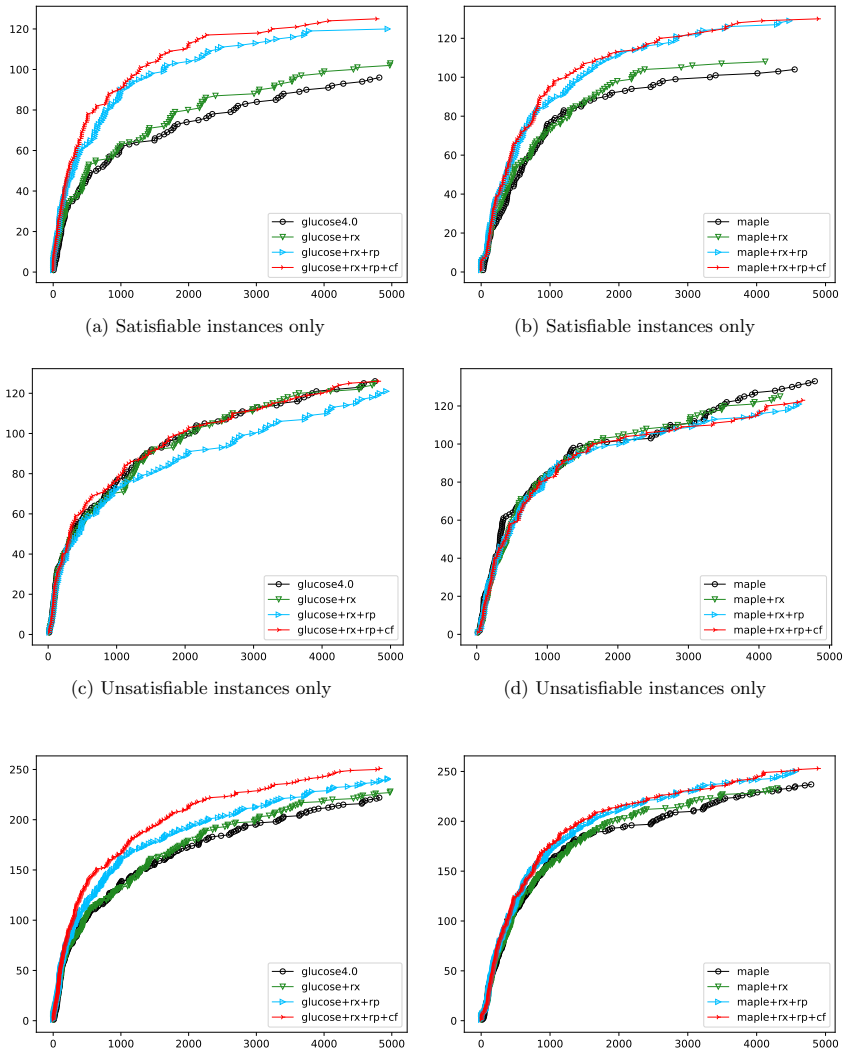
(c) Unsatisfiable instances only

(d) Unsatisfiable instances only

(e) All instances

(f) All instances

Figure 9: CDF for the solvers Glucose (left) and Maple (right) about the relaxed CDCL, local search rephasing and conflict frequency on benchmarks from the SAT Competition 2020



(a) Satisfiable instances only

(b) Satisfiable instances only

(c) Unsatisfiable instances only

(d) Unsatisfiable instances only

(e) All instances

(f) All instances

Figure 10: CDF for the solvers Glucose (left) and Maple (right) about the relaxed CDCL, local search rephasing and conflict frequency on benchmarks from the SAT Competition 2021

References

Anbulagan, Pham, D. N., Slaney, J. K., & Sattar, A. (2005). Old resolution meets modern SLS. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pp. 354–359. AAAI Press / The MIT Press.

Audemard, G., Lagniez, J., Mazure, B., & Sais, L. (2009). Integrating conflict-driven clause learning to local search. In Deville, Y., & Solnon, C. (Eds.), *Proceedings 6th International Workshop on Local Search Techniques in Constraint Satisfaction, LSCS 2009, Lisbon, Portugal, 20 September 2009*, Vol. 5 of *EPTCS*, pp. 55–68.

Audemard, G., Lagniez, J., Mazure, B., & Sais, L. (2010). Boosting local search thanks to CDCL. In Fermüller, C. G., & Voronkov, A. (Eds.), *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, Vol. 6397 of *Lecture Notes in Computer Science*, pp. 474–488. Springer.

Audemard, G., & Simon, L. (2009). Predicting learnt clauses quality in modern SAT solvers. In Boutilier, C. (Ed.), *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pp. 399–404.

Audemard, G., & Simon, L. (2012a). Glucose 2.1: Aggressive—but reactive—clause database management, dynamic restarts. In *Workshop on the Pragmatics of SAT 2012*.

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

Balint, A. (2014). *Engineering stochastic local search for the satisfiability problem.* Ph.D. thesis, University of Ulm.

Balint, A., Belov, A., Järvisalo, M., & Sinz, C. (2015). Overview and analysis of the SAT Challenge 2012 solver competition. *Artificial Intelligence*, *223*, 120–155.

Balint, A., Henn, M., & Gableske, O. (2009). A novel approach to combine a SLS- and a DPLL-solver for the satisfiability problem. In Kullmann, O. (Ed.), *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, Vol. 5584 of *Lecture Notes in Computer Science*, pp. 284–297. Springer.

Balint, A., & Manthey, N. (2018). SparrowToRiss 2018. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*, Vol. B-2018-1 of *Department of Computer Science Report Series B*, pp. 38–39, Finland. Department of Computer Science, University of Helsinki.

Balint, A., & Schöning, U. (2012). Choosing probability distributions for stochastic local search and the role of make versus break. In *Proceedings of SAT 2012*, pp. 16–29.

Barnett, L. A., & Biere, A. (2021). Non-clausal redundancy properties. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, Vol. 12699, pp. 252–272.

Barnett, L. A., Cerna, D. M., & Biere, A. (2020). Covered clauses are not propagation redundant. In Peltier, N., & Sofronie-Stokkermans, V. (Eds.), *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, Vol. 12166 of *Lecture Notes in Computer Science*, pp. 32–47. Springer.

Barrett, C., Sebastiani, R., Seshia, S. A., & Tinelli, C. (2021). Satisfiability modulo theories. In Biere, A., Heule, M. J. H., van Maaren, H., & Walsh, T. (Eds.), *Handbook of Satisfiability* (Second edition)., Vol. 336, pp. 1267–1369. IOS Press.

Biere, A. (2010). Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. In *FMV Report Series Technical Report*, Vol. 10.

Biere, A. (2014). Yet another local search solver and Lingeling and friends entering the SAT competition 2014. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, Vol. 2014, p. 65.

Biere, A. (2017a). CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In Balyo, T., Heule, M. J. H., & Järvisalo, M. (Eds.), *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, Vol. B-2017-1 of *Department of Computer Science Series of Publications B*, pp. 14–15. University of Helsinki.

Biere, A. (2017b). Deep Bound Hardware Model Checking Instances, Quadratic Propagation Benchmarks and Reencoded Factorization Problems Submitted to the SAT Competition 2017. In Balyo, T., Heule, M. J. H., & Järvisalo, M. (Eds.), *Proceedings of SAT Competition 2017 – Solver and Benchmark Descriptions*, Vol. B-2017-1 of *Department of Computer Science Series of Publications B*, pp. 40–41. University of Helsinki.

Biere, A. (2018). CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*, Vol. B-2018-1, pp. 13–14.

Biere, A. (2019). CaDiCaL at the SAT Race 2019. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, Vol. B-2019-1 of *Department of Computer Science Series of Publications B*, pp. 8–9. University of Helsinki.

Biere, A., Fazekas, K., Fleury, M., & Heisinger, M. (2020). CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Competition 2020 – Solver and Benchmark Descriptions*, pp. 51–53.

Biere, A., & Fleury, M. (2020). Chasing target phases. In *Workshop on the Pragmatics of SAT 2020*.

Biere, A., & Fröhlich, A. (2015). Evaluating CDCL variable scoring schemes. In *Theory and Applications of Satisfiability Testing – SAT 2015 – 18th International Conference,*

Austin, TX, USA, September 24-27, 2015, Proceedings, Vol. 9340 of *Lecture Notes in Computer Science*, pp. 405–422. Springer.

Biere, A., & Fröhlich, A. (2015). Evaluating CDCL restart schemes. In Heule, M. J. H., & Weaver, S. (Eds.), *Theory and Applications of Satisfiability Testing—SAT 2015—18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, Vol. 9340 of *Lecture Notes in Computer Science*, pp. 405–422. EasyChair.

Biere, A., Järvisalo, M., & Kiesl, B. (2021). Preprocessing SAT solving (second edition). In Biere, A., Heule, M. J. H., van Maaren, H., & Walsh, T. (Eds.), *Handbook of Satisfiability*, Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 391–435. IOS Press.

Biere, A., Järvisalo, M., Le Berre, D., Meel, K. S., & Mengel, S. (2020). *The SAT Practitioner's Manifesto*.

Biere, A., & Kröning, D. (2018). SAT-based model checking. In *Handbook of Model Checking*, pp. 277–303. Springer.

Cai, S. (2015). *Novel Local Search Methods for Satisfiability*. Ph.D. thesis, Griffith University.

Cai, S., Luo, C., & Su, K. (2014). CCAnr+glucose in SAT Competition 2014. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, No. 2 in Department of Computer Science Series of Publications B, p. 17.

Cai, S., Luo, C., & Su, K. (2015). CCAnr: A configuration checking based local search solver for non-random satisfiability. In Heule, M., & Weaver, S. A. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, Vol. 9340 of *Lecture Notes in Computer Science*, pp. 1–8. Springer.

Cai, S., Luo, C., & Zhang, H. (2017). From decimation to local search and back: A new approach to MaxSAT. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pp. 571–577. ijcai.org.

Cai, S., Luo, C., Zhang, X., & Zhang, J. (2021). Improving local search for structured SAT formulas via unit propagation based construct and cut initialization (short paper). In Michel, L. D. (Ed.), *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, Vol. 210 of *LIPIcs*, pp. 5:1–5:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Cai, S., & Zhang, X. (2018). ReasonLS. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions*, Vol. B-2018-1 of *Department of Computer Science Series of Publications B*, pp. 52–53, Finland. Department of Computer Science, University of Helsinki.

Cai, S., & Zhang, X. (2019). Four relaxed CDCL solvers. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*,

Vol. B-2019-1 of *Department of Computer Science Report Series B*, p. 35, Finland. Department of Computer Science, University of Helsinki.

Cai, S., & Zhang, X. (2021). Deep cooperation of CDCL and local search for SAT. In Li, C.-M., & Manyà, F. (Eds.), *Theory and Applications of Satisfiability Testing – SAT 2021*, pp. 64–81, Cham. Springer International Publishing.

Cha, B., & Iwama, K. (1996). Adding new clauses for faster local search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 1*, pp. 332–337. AAAI Press / The MIT Press.

Eén, N., & Sörensson, N. (2003). An extensible SAT-solver. In Giunchiglia, E., & Tacchella, A. (Eds.), *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, Vol. 2919 of *Lecture Notes in Computer Science*, pp. 502–518. Springer.

Fang, H., & Ruml, W. (2004). Complete local search for propositional satisfiability. In McGuinness, D. L., & Ferguson, G. (Eds.), *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pp. 161–166. AAAI Press / The MIT Press.

Fazekas, K., Biere, A., & Scholl, C. (2019). Incremental inprocessing in SAT solving. In Janota, M., & Lynce, I. (Eds.), *Theory and Applications of Satisfiability Testing – SAT 2019 – 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, Vol. 11628 of *Lecture Notes in Computer Science*, pp. 136–154. Springer.

Gomes, C. P., Selman, B., Crato, N., & Kautz, H. (2000). Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reasoning*, *24*(1/2), 67–100.

Habet, D., Li, C. M., Devendeville, L., & Vasquez, M. (2002). A hybrid approach for SAT. In Hentenryck, P. V. (Ed.), *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, Vol. 2470 of *Lecture Notes in Computer Science*, pp. 172–184. Springer.

Hamadi, Y., Jabbour, S., & Sais, L. (2009). Learning for dynamic subsumption. In *ICTAI 2009, 21st IEEE International Conference on Tools with Artificial Intelligence, Newark, New Jersey, USA, 2-4 November 2009*, pp. 328–335. IEEE Computer Society.

Han, H., & Somenzi, F. (2009). On-the-fly clause improvement. In Kullmann, O. (Ed.), *Theory and Applications of Satisfiability Testing – SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30–July 3, 2009. Proceedings*, Vol. 5584 of *Lecture Notes in Computer Science*, pp. 209–222. Springer.

Heule, M. J. H., Järvisalo, M., & Biere, A. (2013). Revisiting hyper binary resolution. In Gomes, C. P., & Sellmann, M. (Eds.), *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International*

Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings, Vol. 7874 of *Lecture Notes in Computer Science*, pp. 77–93. Springer.

Heule, M. J. H., Kiesl, B., & Biere, A. (2020). Strong extension-free proof systems. *J. Autom. Reasoning*, *64*, 533–554.

Heule, M. J. H., Kullmann, O., & Marek, V. W. (2016). Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In Creignou, N., & Berre, D. L. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, Vol. 9710 of *Lecture Notes in Computer Science*, pp. 228–245. Springer.

Hirsch, E. A., & Kojevnikov, A. (2005). UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Ann. Math. Artif. Intell.*, *43*(1), 91–111.

Hoos, H. H., & Stützle, T. (2004). *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann.

Järvisalo, M., & Biere, A. (2010). Reconstructing solutions after blocked clause elimination. In Strichman, O., & Szeider, S. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, Vol. 6175 of *Lecture Notes in Computer Science*, pp. 340–345. Springer.

Järvisalo, M., Heule, M., & Biere, A. (2012). Inprocessing rules. In Gramlich, B., Miller, D., & Sattler, U. (Eds.), *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, Vol. 7364 of *Lecture Notes in Computer Science*, pp. 355–370. Springer.

Jeroslow, R. G., & Wang, J. (1990). Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, *1*, 167–187.

Kautz, H. A., & Selman, B. (1992). Planning as satisfiability. In Neumann, B. (Ed.), *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings*, pp. 359–363. John Wiley and Sons.

Kiesl, B., Heule, M. J. H., & Biere, A. (2019). Truth assignments as conditional autarkies. In Chen, Y., Cheng, C., & Esparza, J. (Eds.), *Automated Technology for Verification and Analysis – 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*, Vol. 11781 of *Lecture Notes in Computer Science*, pp. 48–64. Springer.

Knuth, D. E. (2006). *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming)*. Addison-Wesley Professional.

Kochemazov, S., Zaikin, O., Kondratiev, V., & Semenov, A. (2019). MapleLCMDistChronoBT-DL, duplicate learnts heuristic-aided solvers at the SAT Race 2019. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*, Vol. B-2019-1 of *Department of Computer Science Report Series B*, pp. 24–24, Finland. Department of Computer Science, University of Helsinki.

Letombe, F., & Marques-Silva, J. (2008). Improvements to hybrid incremental SAT algorithms. In Büning, H. K., & Zhao, X. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, Vol. 4996 of *Lecture Notes in Computer Science*, pp. 168–181. Springer.

Li, C. M., & Habet, D. (2014). Description of RSeq2014. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, Vol. 2014, p. 72.

Li, C. M., & Li, Y. (2012). Satisfying versus falsifying in local search for satisfiability - (poster presentation). In Cimatti, A., & Sebastiani, R. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, Vol. 7317 of *Lecture Notes in Computer Science*, pp. 477–478. Springer.

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

Lorenz, J., & Wörz, F. (2020). On the effect of learned clauses on stochastic local search. In Pulina, L., & Seidl, M. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, Vol. 12178 of *Lecture Notes in Computer Science*, pp. 89–106. Springer.

Luby, M., Sinclair, A., & Zuckerman, D. (1993). Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, *47*(4), 173–180.

Manthey, N. (2010). Improving SAT solvers using state-of-the-art techniques. Master's thesis, Diploma thesis, Institut für Künstliche Intelligenz, Fakultät Informatik.

Marques Silva, J. P., Lynce, I., & Malik, S. (2021). Conflict-drive clause learning SAT solvers. In *Handbook of Satisfiability* (Second edition)., Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 133–182. IOS Press.

Mazure, B., Sais, L., & Grégoire, É. (1998). Boosting complete techniques thanks to local search methods. *Ann. Math. Artif. Intell.*, *22*(3-4), 319–331.

Möhle, S., & Biere, A. (2019). Backing backtracking. In Janota, M., & Lynce, I. (Eds.), *Theory and Applications of Satisfiability Testing – SAT 2019 – 22nd International Conference,*, Vol. 11628 of *Lecture Notes in Computer Science*, pp. 250–266. Springer.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001*, pp. 530–535.

Nadel, A., & Ryvchin, V. (2018). Chronological backtracking. In Beyersdorff, O., & Wintersteiger, C. M. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, Vol. 10929 of *Lecture Notes in Computer Science*, pp. 111–121. Springer.

Newman, N., Fréchette, A., & Leyton-Brown, K. (2018). Deep optimization for spectrum repacking. *Commun. ACM*, *61*(1), 97–104.

Oh, C. (2015). Between SAT and UNSAT: the fundamental difference in CDCL SAT. In Heule, M., & Weaver, S. A. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, Vol. 9340 of *Lecture Notes in Computer Science*, pp. 307–323. Springer.

Pipatsrisawat, K., & Darwiche, A. (2007). A lightweight component caching scheme for satisfiability solvers. In Marques-Silva, J., & Sakallah, K. A. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, Vol. 4501 of *Lecture Notes in Computer Science*, pp. 294–299. Springer.

Prasad, M. R., Biere, A., & Gupta, A. (2005). A survey of recent advances in SAT-based formal verification. *Int. J. Softw. Tools Technol. Transf.*, *7*(2), 156–173.

Ramos, A., van der Tak, P., & Heule, M. J. H. (2011). Between restarts and backjumps. In Sakallah, K. A., & Simon, L. (Eds.), *Theory and Applications of Satisfiability Testing—SAT 2011—14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, Vol. 6695 of *Lecture Notes in Computer Science*, pp. 216–229. Springer.

Ryan, L. (2004). Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University.

Ryvchin, V., & Strichman, O. (2008). Local restarts. In Büning, H. K., & Zhao, X. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, Vol. 4996 of *Lecture Notes in Computer Science*, pp. 271–276. Springer.

Selman, B., Kautz, H. A., & McAllester, D. A. (1997). Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pp. 50–54. Morgan Kaufmann.

Silva, J. P. M., & Sakallah, K. A. (2000). Boolean satisfiability in electronic design automation. In Micheli, G. D. (Ed.), *Proceedings of the 37th Conference on Design Automation, Los Angeles, CA, USA, June 5-9, 2000*, pp. 675–680. ACM.

Soos, M. (2013). Strangenight. In Balint, Adrian Belov, A., Heule, M. J. H., & Järvisalo, M. (Eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, Vol. B-2013-1 of *Department of Computer Science Series of Publications B*, p. 1. University of Helsinki.

Soos, M., & Biere, A. (2019). CryptoMiniSat 5.6 with YalSAT at the SAT Race 2019. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Race 2019 – Solver and Benchmark Descriptions*, Vol. B-2019-1 of *Department of Computer Science Series of Publications B*, pp. 14–15. University of Helsinki.

Soos, M., Nohl, K., & Castelluccia, C. (2009). Extending SAT solvers to cryptographic problems. In Kullmann, O. (Ed.), *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July*

3, 2009. Proceedings, Vol. 5584 of *Lecture Notes in Computer Science*, pp. 244–257. Springer.

Vizel, Y., Weissenbacher, G., & Malik, S. (2015). Boolean satisfiability solvers and their applications in model checking. *Proc. IEEE*, *103*(11), 2021–2035.

Zhang, W., Sun, Z., Zhu, Q., Li, G., Cai, S., Xiong, Y., & Zhang, L. (2020). NLocalSAT: Boosting local search with solution prediction. In Bessiere, C. (Ed.), *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pp. 1177–1183. ijcai.org.

Journal Paper

[1] Armin Biere, Fleury, Mathias, Nils Froleyks, and J.H. Marijn Heule. "The SAT Museum". In: *Proceedings of the 14th Internantional Workshop on Pragmatics of SAT Co-located with the 26th International Conference on Theory and Applicationas of Satisfiability Testing (SAT 2003), Alghero, Italy, July, 4, 2023*. Ed. by Matti Järvisalo and Daniel Le Berre. Vol. 3545. CEUR Workshop Proceedings. I wrote the first version of the paper and did some of the experiments and porting. CEUR-WS.org, 2023, pp. 72–87. URL: http://ceur-ws.org/Vol-3545/paper6.pdf.

The SAT Museum

Armin Biere[1,*], Mathias Fleury[1], Nils Froleyks[2] and Marijn J.H. Heule[3]

[1]University of Freiburg, Germany
[2]Johannes Kepler University Linz, Austria
[3]Carnegie Mellon University, Pittsburgh, PA, USA

Abstract
The virtual SAT Solver Museum is an effort towards preserving historical SAT solvers, by collecting and porting their source code to modern compilers and evaluating them on representative benchmark sets on the same hardware. This allows us to compare historic and modern solvers in the same environment. Our results clearly show a remarkable improvement of SAT solver performance in the last 30 years.

Keywords
Satisfiability, SAT Solvers, SAT Competition

1. Introduction

It has been stated that "No major performance breakthrough [happened in SAT solving] in close to two decades". Notable proponents of this claim are Karem Sakallah at the recent Simons Institute's seminar in 2023 and Joao Marques-Silva during his invited talk at POS 2019 [1, Slide 12 (or 53 of the total number)]. The SAT Museum exists to document the history of SAT solving and to show in contrast to these claims that indeed "SAT solvers are getting faster and faster".

The SAT Museum is curated by two authors of this paper; Armin Biere and Marijn Heule have put considerable effort into collecting and restoring the SAT solvers that have been published since the first SAT competitions more than two decades ago. Some results of this effort have been presented as a lightening talk at POS'20, as well as in the form of a (comparatively) high-impact tweet with preliminary plots for the SAT Competition 2020 benchmarks on Twitter.

Even though a first SAT competition was conducted more than 3 decades ago in 1992 [2], the current regular series of annual SAT competitions was started in 2002 [3] by Laurent Simon and Daniel Le Berre and in most years attracts dozens of SAT solver submissions. The SAT competition provides a fair environment where solvers compete on the same benchmarks and hardware. These competitions have been credited as a main driving force in advancing SAT-solving technology and are a well-recognized show-case with high visibility and impact far beyond the core SAT community.

Each year, the benchmark suite consists of a combination of old and new benchmarks. In recent years, at least 75% of the benchmarks were new, and no more than 14 out of 400 originated

14th International Workshop on Pragmatics of SAT (PoS 2023)
*Corresponding author.
0000-0001-7170-9242 (A. Biere); 0000-0002-1705-3083 (M. Fleury); 0000-0003-3925-3438 (N. Froleyks); 0000-0002-5587-8801 (M. J.H. Heule)

from the same research group to ensure diversity. For more information on the competition we refer to the yearly SAT competition proceedings, e.g., the SAT Competition 2022 proceedings [4], or to the last article describing the SAT Competition in 2020 [5].

To asses the progress in solver performance, we consider all winning solvers since the SAT Competition 2002, whose code we could find on the SAT Competition website or obtain through personal communication. Note, that there was no requirement to publish source code nor even binaries in earlier incarnations of the SAT Competition (Section 3). Besides providing data on competition winners we also include two historically important solvers: Boehm1, the winner of the first SAT competition in 1992 [2], as well as Grasp [6] from 1997.

We run on the same hardware (from 2016) all collected and patched solvers on six benchmark sets from SAT competitions spanning more than two decades, namely 2002, 2011, 2019, 2020, 2021, and 2022. We report results on each set separately, in order to address an argument brought forward by Laurent Simon at a recent POS workshop, that the benchmark selection method of more recent competitions might give a bias towards newer solvers and which arguably might not be observable on the SAT Competition 2011 benchmark set for example. Our data on the SAT Competition 2011 benchmark set refutes this argument as it clearly shows the same solver progress which we observed in other years.

While in general we see a big improvement in solver performance in these 30 years across all considered benchmark sets, the yearly improvement is mostly rather slow, except for performance jumps in some years, which arguably happen with a frequency of 3 to 5 years. Analyzing the reasons for this apparent progress, i.e., both with respect to algorithms, heuristics and implementation, and in particular distilling the core ideas leading to these performance jumps is considered an important follow-up work but out of the scope of this first study.

2. Preliminaries

For the sake of understanding this paper, no special knowledge of SAT is required and we refer to the *Handbook of Satisfiability* [7] for more details. In short, CDCL [8] and its predecessor DPLL [9] work on a partial assignment trying to satisfy a set of clauses. When a *conflict* (mismatch between the assignment and the constraints) arises, the partial model is adapted and CDCL learns new clauses to prevent the same conflict in the future.

On top of CDCL or DPLL, the set of clauses can be simplified by transforming the problem more significantly. In earlier solvers, these techniques were employed as *preprocessing* before running CDCL, whereas nowadays they are run interleaved with CDCL as *inprocessing*. We refer to the corresponding preprocessing chapter [10] in the SAT handbook for details.

In general, this work considers SAT solvers (some of them developed by the first author), which are run on problems from the SAT Competition (last two authors were frequently part of the committee running it and selecting benchmarks).

Therefore, a major thread to validity of this work, as noted by one reviewer, is that its authors are all stake holders in the SAT Competition, either as participants or as organizers. Showing newer solvers to be better clearly serves their interest to support the competition and how its artifacts are used in the scientific discourse on SAT. Nevertheless, we argue, that our carefully executed and extensive experiments are convincing and allow to reach the favorable conclusion,

that SAT solvers are getting faster and faster.

The second major thread to validity is related to the fact that new solvers during development are trained on at-that-time current set of benchmarks: To join the competition, developers check that new technique work on previous competition benchmarks. For example, Kissat-mab-hywalk-2022, the winner of 2022, is based on the 2021 winner (which in turn is based on the 2020 winner). And unsurprisingly, it performs better on the 2021 benchmarks than the 2021 winner. However, none of them has seen the 2022 benchmarks. It is also unlikely that it was trained to perform well on the 2002 benchmarks. To counteract this potential thread to validity, we have used a large benchmark, spanning more than two decades of competitions.

3. The Solvers

In this section, we list all tested solvers attempting to highlight some of their contributions. We selected most SAT Competition winners and some others for their historical significance.

Before Preprocessing. In context of the SAT solver Grasp [6] CDCL was proposed, even though the term CDCL was only later introduced [11]. The decision heuristic at that time attempted to satisfy as many clauses as possible and is considered to be very costly to compute in each search node. Improving such decisions heuristics was also the main topic for the DPLL and thus pre-CDCL SAT solvers participating in the first SAT competition in 1992, from which we include the winning solver Boehm1 [2]. The next historically most significant SAT solver is Chaff [12]. It introduced various techniques that are now commonly used in all SAT solvers, such as watched literals for efficient propagation and the VSIDS decision heuristic to quickly find good decisions. We also consider its 2004 variant [13].

In 2002, the solver Limmat [14] won the competition (by one instance in a tie-breaking round). It follows the ideas of zChaff (at a time when the source code was not available). In 2003 the Siege SAT solver [11] finished 3rd in the SAT Competition (but run hors concours). Its main features are *blocking literals* and the *variable-move-to-front* (VMTF) decision heuristic.

The SAT solver Berkmin [15] improved the Chaff bumping heuristics by more explicitly picking literals in recently learned clauses and also taking into account literals that appear during the conflict analysis and not only those appearing in the final conflict clause. Around this time, restarts were still mostly random. In 2003 the SAT solver MiniSat [16] appeared[1], introducing essential algorithmic and implementation optimizations, including learned clause minimization, exponential VSIDS, and lazy priority queue updates. It won for the first time in 2006. The solver is further considered an attempt at providing clean code by removing redundant features present in other SAT solvers.

CDCL and Preprocessing. In 2005 actually SatElite-GTI, a combination of the SatElite preprocessor [17] with MiniSat as back-end solver, won the competition by a big margin, i.e., contributing to one of those performance jumps we will see. In that year, MiniSat 2005 was also awarded, but it lost to the combination with SatElite. After this success many winning solvers followed this recipe and included SatElite as preprocessor, until in 2008 MiniSat's version 2.0

[1]This seminal paper received the first test-of-time award of the SAT conference in 2022.

won the competition. It is the first MiniSat version which combines CDCL with preprocessing in one code base and executable.

In 2006 MiniSat dominated the (first) SAT Race 2006 and in 2007 the idea of rapid restarts and phase saving helped the Rsat solver to win the SAT Competition 2007. This technique afterwards became standard in all solvers. Also on the CDCL side the fruits of using the glue (LBD) metric [18] of learned clauses to improve reduction of the learned clause data base as well as improved dynamic restart schemes let the Glucose solver [18] win in 2011 and 2012.

The Glucose solver accordingly formed the basis of the development of the MapleSAT solver series winning the competition three times in a row from 2016-2018. In 2016 it introduced the idea of interleaving different policies for SAT (fewer restarts / longer assignments) and UNSAT (more restart / short assignments) proposed by Chanseok Oh [19], contradicting earlier intuitions that restarts mostly help solvers to avoid heavy-tail phenomenon [20] in satisfiable formulas. The solver further included the new LRB decision heuristic and recursive reason side bumping [21] in 2016 [22].

In 2017 vivification [23] was incorporated into MapleSAT in the form of simplifying (aka "inprocessing" - see below) of learned clauses, while before vivification was only applied to original / irredundant clauses during preprocessing. In 2018 the next variant of MapleSAT won the competition, again extended by a different set of authors, by switching between the default CDCL version of non-chronological backtracking and chronological backtracking [24, 25]. In the SAT Race 2019 MapleSAT was again successful by filtering out redundant learned clauses through hashing [26] and enforcing deterministic switches between LRB and VSIDS [27].

Inprocessing Solvers. While the earlier listed solvers did not perform any global transformation on the formula or only do so at the beginning, a different line of work is to include techniques such as probing, subsumption, and blocked clauses during search.

In 2009, the winning SAT solver PrecoSAT [28] implemented this form of formula simplification during search as the first of its kind. This would later be called *inprocessing* [29]. While inprocessing can improve performance, when and for how long to schedule and preempt various inprocessing algorithms becomes both important and difficult to get right.

In 2010, CryptoMiniSat [30] won the competition. It is mainly known for its special handling of XOR clauses (parity or equivalence constraints) which are featured prominently but actually were never used as CryptoMiniSat 2010 could not recover XORs with more than 2 inputs from the CNF. Beyond that CryptoMiniSat features probing and hyper binary resolution in an inprocessing fashion. Initially based on MiniSat, development is still continuing today.

The solver Lingeling (winning 2013 [31] and 2014 [32]) utilized advanced inprocessing techniques, including equivalence reasoning and blocked clause elimination. These developments were enabled by the proper theoretical foundations for model reconstruction [29]. In 2015, the SAT solver abcdSAT [33] won the SAT Race 2014. It uses Lingeling as a preprocessor and Glucose as main solver and featured a new strategy to keep recently used clauses.

Finally, in 2020, the SAT solver Kissat [34] was introduced. Compared to its predecessor CaDiCaL [35], Kissat has fewer features, particularly inprocessors, which however are scheduled more aggressively. While the performance improvement in 2020 is usually attributed to the inclusion of a local search solver and target phases [36], it is worth noting that CaDiCaL was

actually the first solver to explore this. Successors of Kissat won the following two years; in 2021, a version with a more stable decision heuristic [37], and in 2022, an implementation featuring aggressive random walks [38].

4. Collecting and Porting Solvers

In our view the most important outcome of this study is to collect solvers which either in the competition or otherwise are important to the history of SAT solving. Furthermore we ported these legacy solvers to modern compilers. This finally also allowed us to run and compare them, also with more recent solvers, in a clean apple-to-apple comparison on the same hardware and on the same set of benchmarks to asses the progress in the last quarter of a century.

We started this endeavor in 2019, probably right on time, as collecting original solvers is becoming harder than we imagined. Most of the historic solvers still have webpages but links to actual code are dysfunctional. Some webpages disappeared completely. In these cases we reached out to the original authors, which dug through their old computers or found other ways to help us out to retrieve source code or binaries (see acknowledgments at the end).

On top of collecting original source code and binaries, we also provide patches to several legacy solvers, which allow us to compile them with modern compilers (we used gcc/g++ 9.4.0 for our experiments). Most of these issues were due to the g++ compiler becoming over the years more picky about what C++ constructs are accepted. Besides removing some warnings in theses patches, they also contain several fixes addressing bugs of some solvers, which led to incorrect results, but which after debugging were only due to hard coded limits in parsers or in the case of zChaff due to the code not being 64-bit clean. Patches will have to be updated for newer version of the gcc/g++, similar to the restoration process in an ordinary museum.

Besides porting solvers, we also fixed the parser of the solver Boehm1 to support DIMACS and to parse more than 1 000 variables. The implementation of the solver is actually recursive. During experiments we considered increasing the stack size to reduce the number of errors, but finally decided against this option and kept the default stack size (of 8 MB). In the end no solver run showed any discrepancy on the 6 competition benchmark sets we used in the experiments.

5. Performance Results

We ran all the benchmarks on 8-core Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (turbo-mode disabled) with a memory limit of 127 GB and a time limit of 5 000 seconds as in recent SAT competitions even though on slightly slower hardware. All data, including solvers, patches, log files and plots, is available at https://cca.informatik.uni-freiburg.de/satmuseum and [39].

Regarding results we want to stress again that SAT solver developers train on previous competitions: The winner of the 2022 competition is based on the winner from 2021, had access and most likely has been trained on the 2021 benchmarks to find better heuristics than its predecessor from 2021, which has not been trained on them nor on more recent problems sets.

We use CDFs (cumulative distribution function) and not cactus plots: the higher the solver the more problems solved and the more to the left the faster the solver. Our results are shown in Fig. 1 for the SAT Competition 2002, in Fig. 2 for the SAT Competition 2011, in Fig. 3 for the

SAT Competition 2019, in Fig. 4 for the SAT Competition 2020, in Fig. 5 for the SAT Competition 2021, and in Fig. 6 for the SAT Competition 2022. The conclusion is clear: consistently across all benchmarks of several years, recent solvers are better than old solvers with some minor variability. More recent solvers do solve more instances.

Nevertheless, there seems to be several major performance jumps in the reported results: one when preprocessing was introduced around 2006 and a second inconsistent one: 2019 on 2021/2022 benchmarks and 2016 on older benchmark sets with a minor improvement in 2020. The second jump can be attributed to a combination of local search (after 2019 in CaDiCaL and Kissat), rephasing (after 2016), and more aggressive bounded variable elimination.

The behavior of solvers varies across benchmarks. The SAT solver maple-compsps-drup is a striking example: after 2 500 seconds it changes the heuristics to switch to VSIDS (later variants would change on a more regular and deterministic interval). In 2002 the effect is quite strong and it solves many instances in a very short time (while still performing worse than Lingeling 2013). This effect is less pronounced in 2011 and barely visible later.

In Figure 7, we visualize the pairwise similarity between solvers. The highest similarity is observed between the two most recent solvers, Kissat (2021) and Kissat (2022), while the lowest similarity is observed between these two solvers and the oldest solvers, Boehm1 (1992) and Grasp (1997). The dendrogram above the heat-map depicts a hierarchical clustering automatically generated based on solver similarity. Remarkably, it very closely aligns with release years.

Generally, solvers developed after the introduction of a particular technique utilize that technique and gain an advantage on the same benchmarks. This is most evident with the three main clusters: the first groups solvers before introduction of preprocessing, and the other two clusters group solvers before and after 2016. In that year, a new heuristic was introduced and specialized phases targeting SAT and UNSAT problems became popular. The split between the early preprocessing solvers and the inprocessing solvers after 2009 is also easily discernible.

Related Work. In this work, we compare SAT solvers on SAT competition instances. In related work Dutertre compared various SAT solvers from 2019 (without restricting to competition winners) and MiniSat but on SMT bitvector benchmarks [40]. He observed improvements over MiniSat but the ranking did not reflect the results from the SAT Competition – the second best SAT solver finished 7th. He also tested various features in CaDiCaL and tested on problems hard for SAT solvers not requiring extensive theory reasoning. Recent work by Fazekas [41] on simplifying the interface between SAT and SMT solvers reaches similar conclusions.

In another related work by Kochemazov, Ignatiev, and Marques-Silva [42] the focus was on *incremental* SAT solving. They compared competition winners between 2016 and 2020 and MiniSat, but they did not observe an improvement over MiniSat on MaxSAT instances, except for two families. However, in recent work on incremental use of SAT solvers for backbone extraction [43] the more recent solver CaDiCaL surpasses MiniSat by a large margin.

The SAT heritage effort by Audemard, Paulevé, and Simon [44] also tries to preserve and enable to run historic SAT solvers. Their approach is based on system-level virtualization with docker containers and thus orthogonal to ours by compiling the original source code with historic compilers. They do not attempt to port and patch solvers, which means a comparison such as ours on 6 sets of competition benchmark sets will result in many discrepancies, particularly for

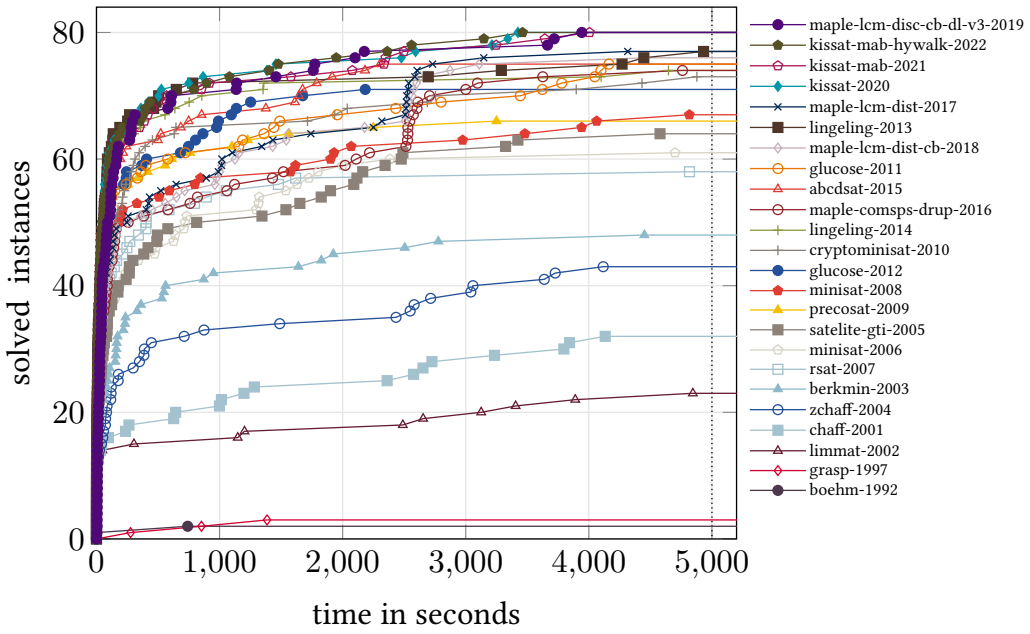


Figure 1: All time winners on the SAT Competition 2002 benchmarks (100 problems)

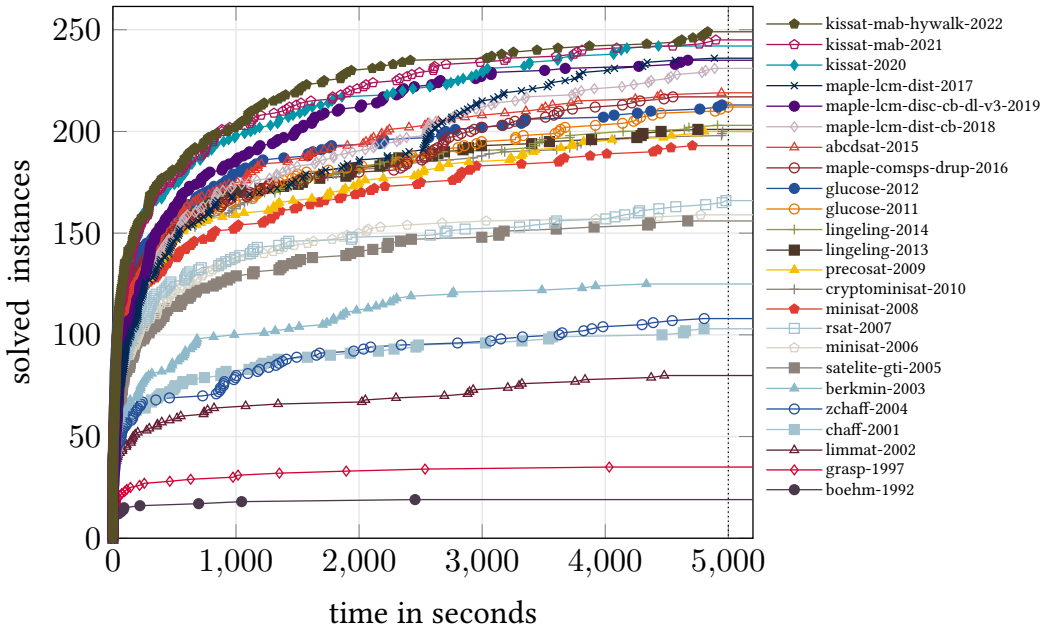


Figure 2: All time winners on the SAT Competition 2011 benchmarks (300 problems)

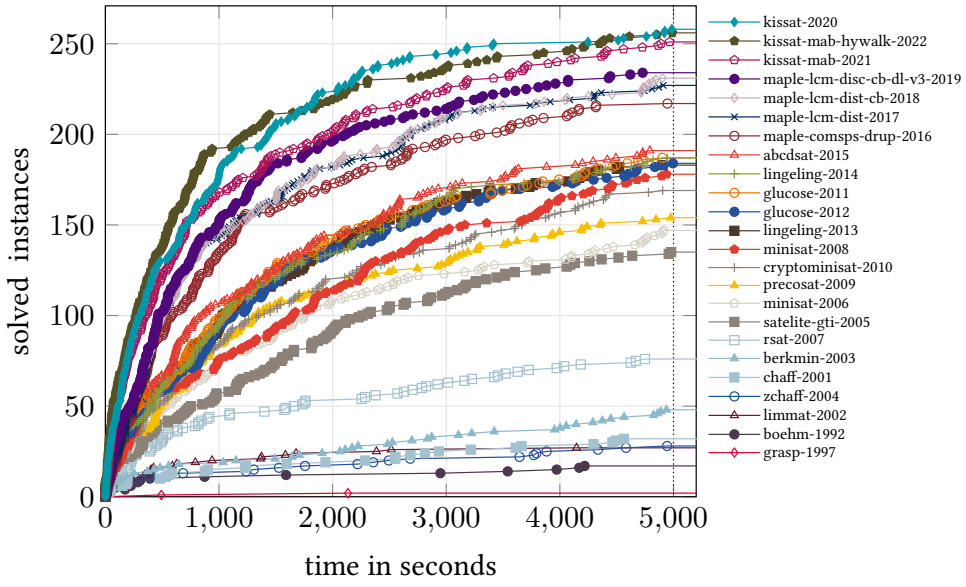


Figure 3: All time winners on the SAT Competition 2019 benchmarks (400 problems)

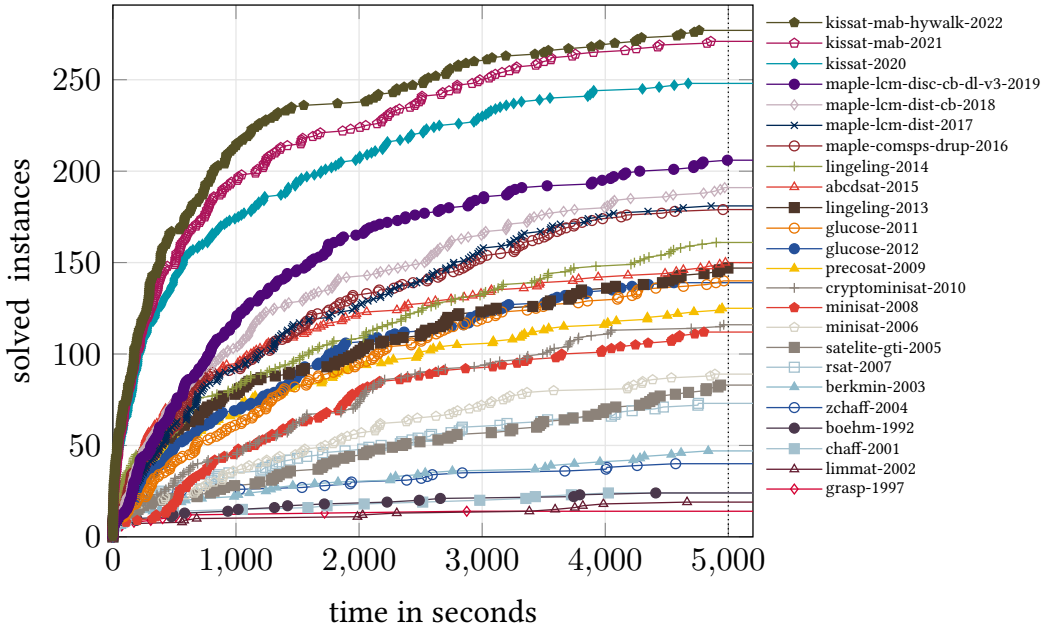


Figure 4: All time winners on the SAT Competition 2020 benchmarks (400 problems)

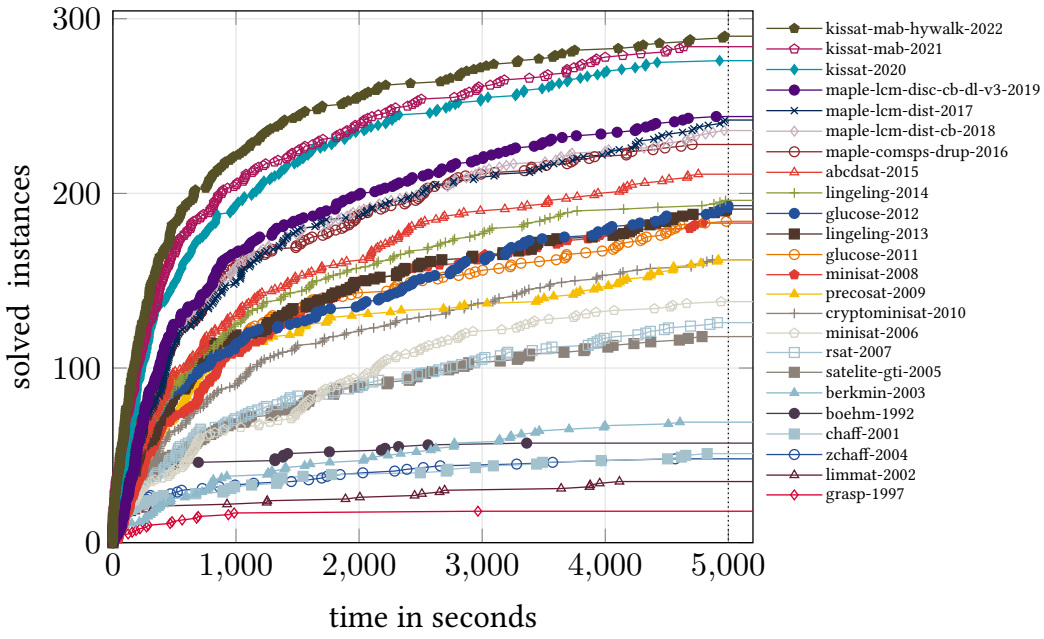


Figure 5: All time winners on the SAT Competition 2021 benchmarks (400 problems)

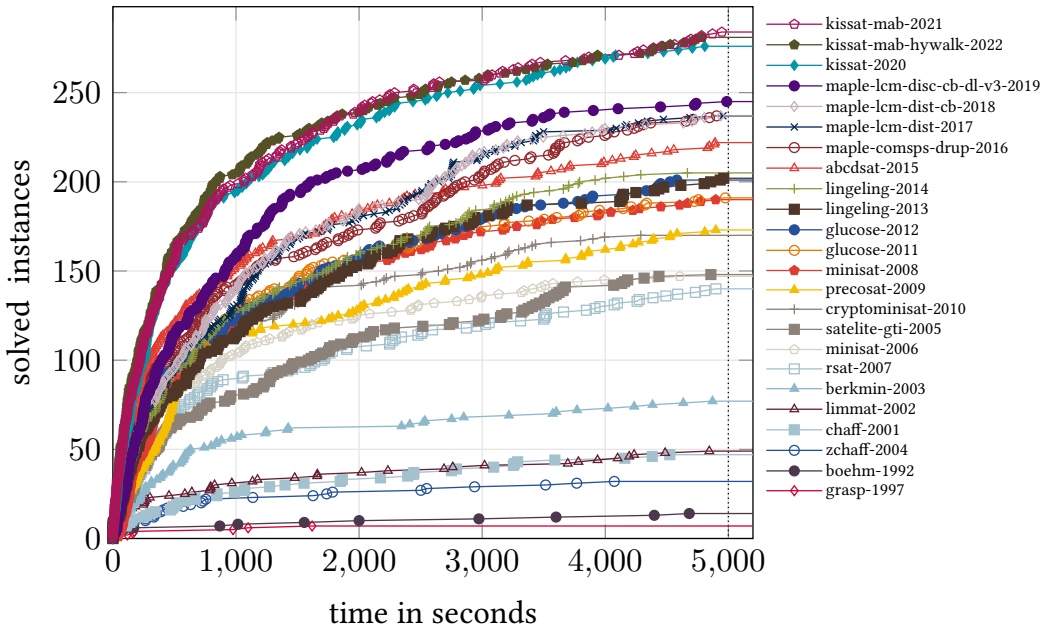


Figure 6: All time winners on the SAT Competition 2022 benchmarks (400 problems)

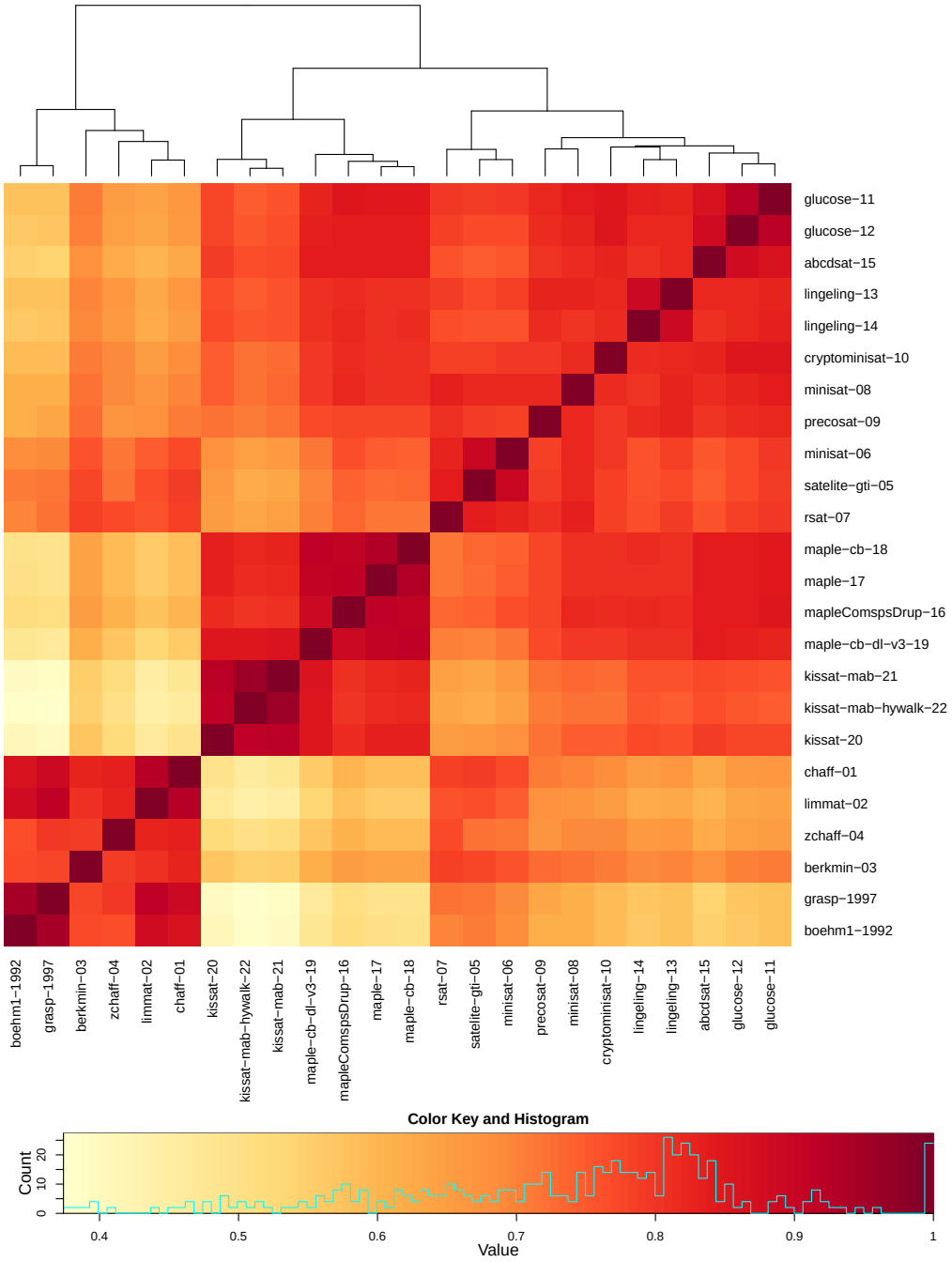


Figure 7: Heat-map and dendrogram (top) based on runtime similarity. The similarity between two solvers is defined by comparing the solving time they achieve on each of the 2000 benchmarks over the years. If a solver failed to solve an instance, we assign twice the timeout value (10 000). The absolute difference in solving time is then accumulated and normalized to the interval $[0, 1]$, where 1 indicates identical performance ($1 - \sum |t_i - t'_i| \ / \ 2000 \cdot 10\,000$). Darker regions indicate higher similarity between solvers. A more precise relation between color and similarity-value together with a histogram of the values that appear is given at the bottom. Above the heat-map we illustrate a hierarchical clustering, with solvers or clusters with high similarity join lower in the dendrogram, while clusters with significantly different performance are joined higher.

older solvers, and thus render it meaningless from the perspective of comparing performance. Our approach will likely need additional patches for newer compilers in the future though. However it is unclear whether container virtualization can survive decades without maintenance.

Finally, an on-first-sight related but in our view bogus experiment was conducted in 2020 by Fichte, Hecher, and Szeider [45]. It only focused on a small rather uncommon benchmark set [46] of 202 benchmarks as well as on a small set of solvers. The key feature of their set-up was to run unmodified legacy solver code on old legacy hardware (from 1999) as well as modern hardware (from 2019), with the goal to compare SAT solver progress due to algorithms / software (team SW) versus progress due to hardware improvements (team HW).

However, we argue that this goal was not reached, as the experiment ignored apparent discrepancies: If we take for example the problem `AProVE07-04.cnf` from the SAT Competition 2012, zChaff claims that this problem is SAT (without having made any decision) within 0.1 s, while all other solvers report UNSAT (as expected). We further observed 7 discrepancies for team SW, on the old Sparc architecture and 9 for team HW on the new architecture, not including problems solved only by zChaff. This actually changes the result substantially and makes the team SW look much better than what was reported in [45]. Amazingly, the solver Grasp also (incorrectly) claims that `AProVE07-04.cnf` is satisfiable. After pointing out these issues to the authors they promised to address these problems in an extended version of their paper.

6. Conclusion

In this work we have collected and fixed the source code of winning SAT solvers from the SAT competitions and compared their performance on many benchmarks. Overall more recent solvers solve more problems, rather consistently. Thus SAT solvers get faster and faster. We are looking forward to continue this preservation effort and performance evaluation on additional older and future solvers as well as benchmark sets.

Clearly, our presented results disprove the false view discussed in the introduction that there was no major progress in SAT solving in the last 20 years. Still, one nagging remaining issue with our work is that we do not provide a deeper understanding about the differences between solvers and whether all implemented techniques are useful. Are there some old techniques not part of modern SAT solvers which are still useful? And most important, can we produce even better SAT solvers by understanding this remarkable progress better?

Acknowledgments

This work is supported by the Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), the LIT AI Lab funded by the State of Upper Austria, and the National Science Foundation under grant CCF-2229099. We further like to thank Daniel Le Berre for the original idea of conducting this study and also coming up with the name “SAT Museum” and are also grateful to the anonymous reviewers for their positive and very encouraging feedback.

We are also in debt to the colleagues who helped us to resurrect the source code or binaries of some of the discussed historic solvers, including Jingchao Chen (for the binary abcdSAT 2015 as the competition did not keep binaries), Eugene Goldberg and Yakov Novikov (for helping us

to retrieve berkmin 2003). Lintao Zhang, Matthew Moskewicz and Sharad Malik (for helping us to revive Chaff2), Hans Kleine Büning and Theodor Lettmann (for forwarding source code of the winning solver Boehm1 in 1992 by Max Böhm) and last but not least João Marques-Silva for providing a patched version of Grasp.

Finally we want to thank the authors of all the other solvers, the participants and organizers of past SAT competitions. Without their enthusiasm in supporting the competition and the community this study would not exist. Last but not least we want to thank Karem Sakallah for his infinite patience in related discussions on the progress in SAT solving and the participants of past editions of the POS workshop.

References

[1] J. P. Marques-Silva, SAT: Disruption, demise & resurgence, 2019. URL: http://www.pragmaticsofssat.org/2019/disruption.pdf, invited talk.

[2] M. Buro, H. Kleine Büning, Report on a SAT competition, Fachbereich Math.-Informatik, Univ. Gesamthochschule Paderborn, Germany, 1992.

[3] L. Simon, D. L. Berre, E. A. Hirsch, The SAT2002 competition, Ann. Math. Artif. Intell. 43 (2005) 307–342. doi:10.1007/s10472-005-0424-6.

[4] T. Balyo, M. J. H. Heule, M. Iser, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions, volume B-2022-1 of *Department of Computer Science Series of Publications B*, University of Helsinki, 2022.

[5] N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, M. Suda, Sat competition 2020, Artificial Intelligence 301 (2021) 103572.

[6] J. P. Marques-Silva, K. A. Sakallah, GRASP - a new search algorithm for satisfiability, in: R. A. Rutenbar, R. H. J. M. Otten (Eds.), Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996, IEEE Computer Society / ACM, 1996, pp. 220–227. doi:10.1109/ICCAD.1996.569607.

[7] A. Biere, M. J. H. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, volume 336 of *Frontiers in Artificial Intelligence and Applications*, 2nd ed., IOS Press, 2021. URL: https://doi.org/10.3233/FAIA336. doi:10.3233/FAIA336.

[8] J. P. Marques-Silva, I. Lynce, S. Malik, Conflict-driven clause learning SAT solvers, in: A. Biere, M. J. H. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, volume 336 of *Frontiers in Artificial Intelligence and Applications*, 2nd ed., IOS Press, 2021, pp. 133–182. doi:10.3233/FAIA200987.

[9] M. Davis, G. Logemann, D. W. Loveland, A machine program for theorem-proving, Commun. ACM 5 (1962) 394–397.

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

[11] L. Ryan, Efficient algorithms for clause-learning SAT solvers, Master's thesis, Simon Fraser Univ., 2004. URL: https://www.cs.sfu.ca/~mitchell/papers/ryan-thesis.ps.

[12] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an

efficient SAT solver, in: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001, ACM, 2001, pp. 530–535. doi:10.1145/378239.379017.

[13] Y. S. Mahajan, Z. Fu, S. Malik, Zchaff 2004: An efficient SAT solver, in: H. H. Hoos, D. G. Mitchell (Eds.), Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers, volume 3542 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 360–375. doi:10.1007/11527695_27.

[14] A. Biere, The Evolution from LIMMAT to NANOSAT, Report, ETH Zürich, 2004-04. doi:10.3929/ethz-a-006744011, technical Reports D-INFK.

[15] E. I. Goldberg, Y. Novikov, Berkmin: A fast and robust SAT-solver, in: 2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France, IEEE Computer Society, 2002, pp. 142–149. doi:10.1109/DATE.2002.998262.

[16] N. Eén, N. Sörensson, An extensible SAT-solver, in: E. Giunchiglia, A. Tacchella (Eds.), Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers, volume 2919 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 502–518. doi:10.1007/978-3-540-24605-3_37.

[17] N. Eén, A. Biere, Effective preprocessing in SAT through variable and clause elimination, in: F. Bacchus, T. Walsh (Eds.), Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings, volume 3569 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 61–75. doi:10.1007/11499107_5.

[18] G. Audemard, L. Simon, Predicting learnt clauses quality in modern SAT solvers, in: C. Boutilier (Ed.), IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009, 2009, pp. 399–404.

[19] 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, 2015, pp. 307–323. doi:10.1007/978-3-319-24318-4_23.

[20] C. P. Gomes, B. Selman, N. Crato, H. A. Kautz, Heavy-tailed phenomena in satisfiability and constraint satisfaction problems, J. Autom. Reason. 24 (2000) 67–100. doi:10.1023/A:1006314320276.

[21] J. H. Liang, V. Ganesh, P. Poupart, K. Czarnecki, Learning rate based branching heuristic for SAT solvers, in: N. Creignou, D. L. Berre (Eds.), 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*, Springer, 2016, pp. 123–140. doi:10.1007/978-3-319-40970-2_9.

[22] C. Oh, COMiniSatPS the chandrasekhar limit and GHackCOMSPS, in: T. Balyo, N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions, volume B-2016-1 of *Department of Computer Science Report Series B*, University of Helsinki, 2016, p. 29.

[23] M. Luo, C. Li, F. Xiao, F. Manyà, Z. Lü, An effective learnt clause minimization approach

for CDCL SAT solvers, in: C. Sierra (Ed.), Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, ijcai.org, 2017, pp. 703–711. doi:10.24963/ijcai.2017/98.

[24] A. Nadel, V. Ryvchin, Chronological backtracking, in: O. Beyersdorff, C. M. Wintersteiger (Eds.), Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings, volume 10929 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 111–121. doi:10.1007/978-3-319-94144-8_7.

[25] V. Ryvchin, A. Nadel, Maple_LCM_Dist_ChronoBT: Featuring chronological backtracking, in: T. Balyo, N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions, volume B-2018-1 of *Department of Computer Science Report Series B*, University of Helsinki, 2018, p. 29.

[26] S. Kochemazov, O. Zaikin, A. A. Semenov, V. Kondratiev, Speeding up CDCL inference with duplicate learnt clauses, in: G. D. Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, J. Lang (Eds.), ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020), volume 325 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2020, pp. 339–346. doi:10.3233/FAIA200111.

[27] S. Kochemazov, O. Zaikin, V. Kondratiev, A. Semenov, MapleLCMDistChronoBT-DL, duplicate learnts heuristic-aided solvers at the SAT Race 2019, in: T. Balyo, N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Race 2019 – Solver and Benchmark Descriptions, volume B-2019-1 of *Department of Computer Science Report Series B*, University of Helsinki, 2019, p. 24.

[28] A. Biere, Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010, Technical Report 10/1, Johannes Kepler University Linz, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2010. doi:10.350/fmvtr.2010-1.

[29] M. Järvisalo, M. J. H. Heule, A. Biere, Inprocessing rules, in: B. Gramlich, D. Miller, U. Sattler (Eds.), Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings, volume 7364 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 355–370. doi:10.1007/978-3-642-31365-3_28.

[30] M. Soos, K. Nohl, C. Castelluccia, Extending SAT solvers to cryptographic problems, in: O. Kullmann (Ed.), Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings, volume 5584 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 244–257. doi:10.1007/978-3-642-02777-2_24.

[31] A. Biere, Lingeling, Plingeling and Treengeling entering the SAT Competition 2013, in: A. Balint, A. Belov, M. J. H. Heule, M. Järvisalo (Eds.), Proc. of SAT Competition 2014 – Solver and Benchmark Descriptions, volume B-2013-1 of *Department of Computer Science Series of Publications B*, University of Helsinki, 2013, pp. 51–51.

[32] A. Biere, Yet another local search solver and Lingeling and friends entering the SAT Competition 2014, in: A. Balint, A. Belov, M. J. H. Heule, M. Järvisalo (Eds.), Proc. of SAT Competition 2014 – Solver and Benchmark Descriptions, volume B-2014-2 of *Department*

of Computer Science Series of Publications B, University of Helsinki, 2014, pp. 39–40.

[33] J. Chen, Minisat_bcd and abcdsat: Solvers based on blocked clause decom- position, in: A. Balint, A. Belov, M. J. H. Heule, M. Järvisalo (Eds.), Proc. of SAT Competition 2015 – Solver and Benchmark Descriptions, volume B-2015-1 of *Department of Computer Science Series of Publications B*, University of Helsinki, 2015, pp. 51–51.

[34] A. Biere, K. Fazekas, M. Fleury, M. Heisinger, CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020, in: T. Balyo, N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions, volume B-2020-1 of *Department of Computer Science Report Series B*, University of Helsinki, 2020, p. 51–53.

[35] A. Biere, CaDiCaL at the SAT Race 2019, in: M. J. H. Heule, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Race 2019 – Solver and Benchmark Descriptions, volume B-2019-1 of *Department of Computer Science Series of Publications B*, University of Helsinki, 2019, pp. 8–9.

[36] S. Cai, X. Zhang, M. Fleury, A. Biere, Better decision heuristics in CDCL through local search and target phases, J. Artif. Intell. Res. 74 (2022) 1515–1563. doi:10.1613/jair.1.13666.

[37] M. S. Cherif, D. Habet, C. Terrioux, Kissat_MAB: Combining VSIDS and CHB through multi-armed bandit, in: T. Balyo, N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Competition 2021 – Solver and Benchmark Descriptions, volume B-2021-1 of *Department of Computer Science Report Series B*, University of Helsinki, 2021, p. 15–16.

[38] J. Zheng, K. He, Z. Chen, J. Zhou, C.-M. Li, Combining hybrid walking strategy with Kissat_MAB, CaDiCaL, and LStech-Mapl, in: T. Balyo, N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions, volume B-2022-1 of *Department of Computer Science Report Series B*, University of Helsinki, 2022, p. 20–21.

[39] A. Biere, M. Fleury, N. Froleyks, M. Heule, The SAT Museum POS'23 artifact, 2023. doi:10.5281/zenodo.10037737.

[40] B. Dutertre, An empirical evaluation of SAT solvers on bit-vector problems, in: F. Bobot, T. Weber (Eds.), Proceedings of the 18th International Workshop on Satisfiability Modulo Theories co-located with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Online (initially located in Paris, France), July 5-6, 2020, volume 2854 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020, pp. 15–25.

[41] K. Fazekas, A. Niemetz, M. Preiner, M. Kirchweger, S. Szeider, A. Biere, IPASIR-UP: user propagators for CDCL, in: M. Mahajan, F. Slivovsky (Eds.), 26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy, volume 271 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 8:1–8:13. URL: https://doi.org/10.4230/LIPIcs.SAT.2023.8. doi:10.4230/LIPIcs.SAT.2023.8.

[42] S. Kochemazov, A. Ignatiev, J. P. Marques-Silva, Assessing progress in SAT solvers through the lens of incremental SAT, in: C. Li, F. Manyà (Eds.), Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings, volume 12831 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 280–298. doi:10.1007/978-3-030-80223-3_20.

[43] A. Biere, N. Froleyks, W. Wang, Cadiback: Extracting backbones with cadical, in:

M. Mahajan, F. Slivovsky (Eds.), 26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy, volume 271 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 3:1–3:12. URL: https://doi.org/10.4230/LIPIcs.SAT.2023.3. doi:10.4230/LIPIcs.SAT.2023.3.

[44] G. Audemard, L. Paulevé, L. Simon, SAT heritage: A community-driven effort for archiving, building and running more than thousand SAT solvers, in: L. Pulina, M. Seidl (Eds.), Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings, volume 12178 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 107–113. doi:10.1007/978-3-030-51825-7_8.

[45] J. K. Fichte, M. Hecher, S. Szeider, A time leap challenge for SAT-solving, in: H. Simonis (Ed.), Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings, volume 12333 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 267–285. doi:10.1007/978-3-030-58475-7_16.

[46] H. H. Hoos, B. Kaufmann, T. Schaub, M. Schneider, Robust benchmark set selection for boolean constraint solvers, in: LION, volume 7997 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 138–152.

8 Verified SAT Solving

Conference Paper

[1] Fleury, Mathias and Peter Lammich. "A more Pragmatic CDCL for IsaSAT and targetting LLVM". In: *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*. Ed. by Brigitte Pientka and Cesare Tinelli. I did most of the formalization and the paper and Peter helped in the initial porting. Springer, 2023, pp. 207–219. URL: https://m-fleury.github.io/ox-hugo/FleuryLammich-CADE29.pdf.

A more Pragmatic CDCL for IsaSAT and targetting LLVM (Short Paper)

Mathias Fleury✉ [1,2,3] and Peter Lammich [4,5]

[1] University Freiburg, Freiburg, Germany
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[3] Johannes Kepler University Linz, Linz, Austria
[4] University of Manchester, Manchester, UK
[5] University of Twente, Twente, Netherlands

Abstract. IsaSAT is the most advanced verified SAT solver, but it did not yet feature inprocessing (to simplify and strengthen clauses). In order to improve performance, we enriched the base calculus to not only do CDCL but also inprocess clauses. We also replaced the target of our code synthesis by Isabelle/LLVM. With these improvements, we can solve 4 times more SAT Competition 2022 problems than the original IsaSAT version, and 4.5 times more problems than any other verified SAT solver we are aware of. Additionally, our changes significantly reduce the trusted code base of our verification.

1 Introduction

SAT solving is a very important tool that has been extensively used in various applications like mathematics or cryptography. To ensure the correctness of the answer provided by a SAT solver, there are two approaches: either producing a certificate that can be checked independently or verifying a SAT solver. The first approach has been extensively studied and works very well in practice [19,26,28] – only checked proofs are counted in the SAT Competition [2].

The second approach, i.e., verifying a whole SAT solver is orders of magnitudes more complex than checking a certificate. To this end, the goal of the IsaFoL (Isabelle Formalization of Logic) [3] effort is to develop methodology and libraries for formalizing modern research in automated reasoning. In this context, we have verified a CDCL calculus (conflict-driven clause learning) and a two-watched literals data structure (Sect. 2). To show that they are useful, we have developed the verified SAT solver IsaSAT [8], which we later optimized [12]. To our surprise, it won the EDA Challenge 2021 defeating all the non-verified solvers, but, as expected, it finished last at the SAT Competition 2022 [2]. However, the former used a much shorter timeout (200 s, not announced before the competition) whereas the latter uses 5000 s.

In this paper, we present our new developments in IsaSAT, which make our solver arguably the most advanced formally verified SAT solver to date: inprocessing and verifying fast LLVM code [20] rather than slow functional code.

Inprocessing is a critical feature of modern SAT solvers (e.g., every winner of the SAT Competition since 2013 includes it). In order to use it in our formally verified solver, we had to extend our verified CDCL calculus: Our new PCDCL calculus includes features to encompass various inprocessing techniques, even if we have not yet implemented every possible technique (Sect. 3).

We generate IsaSAT by exporting a model in the interactive theorem prover Isabelle [22] to executable code. Earlier we used Isabelle's default code generator to export to Standard ML (SML). However, the performance was not sufficient – especially memory consumption was very high. Thus, we switched to Isabelle/LLVM [18], which generates LLVM intermediate representation (LLVM IR). Apart from allowing faster imperative code, it also reduced the trusted code base (Sect. 4), replacing the rather niche MLton [27] compiler by only the backend of the widely used LLVM.

Porting our entire development to Isabelle/LLVM required some changes and some cleanup. Moreover, when we implemented and verified inprocessing, we realized that some design decisions need to be improved. In Sect. 5, we report on our experiences and lessons learned while porting and extending IsaSAT.

Finally, we have benchmarked IsaSAT on the problems from the SAT Competition 2022. We show that just porting IsaSAT from SML to Isabelle/LLVM significantly improved the performance, and the new inprocessing techniques combined with heuristic improvements give us another significant increase, demonstrating the usefulness of our PCDCL calculus (Sect. 6).

This presentation is an extended version of our (non-peer-reviewed) system description from the EDA Challenge 2021 [13] and the SAT Competition 2022 [6]. Compared to that version, we have provided much more details on PCDCL, our experience porting the development to LLVM, and performance tests.

2 Preliminaries

CDCL. CDCL is a procedure that builds a partial assignment called *a trail* either by guessing (called *deciding*) or propagating information. If the partial assignment is a model, the algorithm stops. If there is a conflict between the partial assignment and a clause, the partial assignment is repaired and a new clause is learned. For more details (beyond the scope of this paper), we refer the reader to the *Handbook of Satisfiability* [7].

We use a transition system for our formalization of CDCL [8]. Its state consists of the trail M, the (multi)sets of initial and learned clauses (N and U), and the conflict clause to analyze (or None if there is none). We show one rule, `decide`, that adds L to the current assignment M:

```
inductive decide :: 'st ⇒ 'st ⇒ bool where
  undefined_lit M L ⟹ |L| ∈ |N| ⟹
  decide (M, N, U, None) (L · M, N, U, None)
```

If no conflict has been found so far (`None`), we add the new literal L at the beginning of the trail M. We prove that our set of rules is terminating and correct [8].

Code Synthesis. To generate the IsaSAT code, we start from the abstract rules like `decide` and gradually refine it to some deterministic functions using the Refinement Framework [16]. Then, we rely on Sepref [17] to synthesize code: It takes an (Isabelle) function and synthesizes a new version, replacing functional data structures (like lists) by imperative data structures (like arrays). There are two versions of the tool. The older version, which we used before [8, 12], uses Imperative HOL [9] and Isabelle's standard (trusted) code generator [14] to export code into various functional languages. We used Standard ML (SML) with the compiler MLton [27], because it offers (by far) the best performance for our use case. The new Sepref is part of the Isabelle/LLVM library (developed by the second author) and generates LLVM IR from a model of LLVM IR inside the theorem prover. The code generator interprets a shallow embedding of Isabelle/LLVM as equivalent to similar looking LLVM code. This reduces the trusted code base in two ways: first, the trusted pretty printer is simpler, and, second, instead of the rather niche full compiler MLton, we use only the backend of the widely used LLVM [20].

The biggest difference is that Imperative HOL allows arbitrary large arrays and integers, whereas Isabelle/LLVM is more realistic, requiring integers (in particular array offsets, see Sect. 5.1) to have a fixed bit-width.

Related Work. Our goal is to produce a fully verified SAT solver, without any runtime checks, that both *terminates* and returns a *correct model* while using efficient data structures. No other solver achieves all three goals. The SAT solver TrueSAT from Andrici and Ciobaca [1] relies on the original DPLL and uses less efficient data structures (including counters instead of watch lists), but it terminates. Historically, this would roughly correspond to SAT solver from the early 90s. However, it only uses stateless heuristics, and it is not clear if the approach can be extended to CDCL (where the solver learns and keeps new clauses) or to stateful heuristics (like VSIDS [21]). The solvers VERSAT [23] and Creusat [25] go into a similar direction with CDCL instead of DPLL, but prove a weaker correctness property: they only show that an UNSAT result is correct, while a SAT result requires an additional check. Also, termination is not proved. Only proving this partial property makes many proofs considerably easier, in particular adding restarts. Oe et al's solver VERSAT [23] was the first partially verified solver that could run benchmarks from the SAT Competition. More recently, Skotåm [25] has verified in his Master's thesis the SAT solver CreuSAT using the Creusot framework (relying on Why3 internally). While CreuSAT is much faster than VERSAT in our tests, its correctness relies on (trusted) SMT solvers, and the proofs are not checked by a small kernel like our Isabelle code. However, the verification also takes much less time (a few minutes compared to several hours).

Modern SAT solvers use inprocessing to make the subsequent CDCL run heuristically faster [15]. In particular, clauses are strengthened and global transformation (e.g., to remove variables) are applied. Two techniques (that we do not support), variable elimination and addition, slowly change the models of the formula by changing the set of variable. The SAT solver then reconstructs a model

of the original formula at the end. Fazekas et al. [11] made it compatible with incremental SAT solving. All others inprocessing technique fit into our extended CDCL described in the next section.

3 Pragmatic CDCL for Inprocessing

SAT solvers nowadays apply a combination of CDCL (most of the time) and inprocessing (sometimes). Therefore, we extended our calculus similarly. At the core, we have our terminating CDCL. We also allow for formula transformation and restarts. We call the combination *pragmatic CDCL* or PCDCL.

Splitting the Clause Set. Inprocessing makes it possible to strengthen and simplify clauses. However, we want models from the final set of clauses to remain models from the initial set of clauses. Deleting clauses is not possible: if we start with the clauses $A \vee C$ and $B \vee \neg B$, removing the tautology means that the model A of $A \vee C$ is not a model of the initial clause set anymore. Hence we want to keep the literal B without considering the tautology for propagation/conflict.

To solve the issue we split our set of clauses into two parts: clauses that are useful for propagation and clauses that can be ignored but are kept for their literals. Thus we keep the set of all literals $\mathcal{A}$ constant. For our proof of refinement to the original CDCL, we have to make sure that the new behavior is also possible in the original calculus – in particular we do not miss propagations or conflicts. In the case of tautologies, this is simple (they are never used). If we consider subsumption, like $A \vee B$ subsumes $A \vee B \vee C$, whenever the latter propagates, then the former is a conflict. Therefore, the behavior is compatible.

While the idea of splitting our clauses seems surprising, the additional clause sets are only required for the connection to our CDCL transition system, and we entirely remove them when generating the code. Moreover, the refinement is easier as we do not have to update our heuristics to remove literals (and potentially shorten arrays). Finally, this is similar to the behavior of SAT solvers like Kissat [4]: while the clauses are removed, all literals of the problem are set.

In our original refinement, we have split the clauses to distinguish between clauses of length 1 (where we cannot distinguish two distinct literals and thus they cannot fit into our two-watched literals data structures) and longer clauses, but the aim was only distinguishing on the length.

One important point to notice is that the role of our clause sets changes. In our original CDCL, N was the (immutable) set of initial clauses and U contains the redundant clauses that can be removed at any point: N ensures that we do gain new models during our transformations. Now, the set changes: strengthening an irredundant clause from N also shortens the clause that is in there. Therefore, a naive version could remove literals.

Overall we have 4 sets of clauses: the irredundant clauses N and the redundant U clauses, and each one is divided into the active clauses (N_a and U_a) and the inactive (discarded) clauses (N_d and U_d). For example, tautologies or subsumed clauses are discarded, but remain in N, so literals are never removed. In

our development there are actually three sets (containing a literal set at level 0 or tautologies, subsumed clauses, and false clauses) to reduce the number of case distinction in some proofs. We never demote irredundant clauses to redundant ones, but we can promote them.

Inprocessing Rules. Our aim when picking the rule is to be general (like we can learn any useful clause) and then we specialize rules to specific techniques. We will show this with the example of subsumption-resolution [7]. When doing subsumption-resolution, we resolve two clauses together if the conclusion is shorter. Then we can remove either one or both of the antecedents. For example, resolving $A \vee B \vee C$ with $A \vee \neg C$ produces the clause $A \vee B$ with subsumes the former clause. If the latter clause was $A \vee B \vee \neg C$, the resolved clauses would actually subsume both clauses.

One of the most important inprocessing rule learns any possible clause. To simplify the presentation, we will only give the rules operating on the learned clauses, but similar rules exists for the initial set of clauses.

$$\begin{aligned}&\texttt{inductive}\ \mathsf{cdcl\_learn\_clause} :: \textit{'prag\_ st} \Rightarrow \textit{'prag\_ st} \Rightarrow \textit{bool}\ \texttt{where}\\ &\quad |C| \subseteq |N + N_d| \Longrightarrow \mathsf{count\_decided}\, M = 0 \Longrightarrow\\ &\quad N \wedge N_d \models C \Longrightarrow \neg\mathsf{tautology}\, C \Longrightarrow \mathsf{distinct}\, C \Longrightarrow\\ &\quad \mathsf{cdcl\_subsumed}\ (M, N, U, \mathsf{None}, N_d, U_d)\\ &\qquad\qquad (M, N, U \wedge C, \mathsf{None}, N_d, U_d)\end{aligned}$$

The side conditions not only include that the clause is entailed and duplicate-free, but also the clause is not a tautology and we do not break CDCL invariants ($\mathsf{count\_decided}\, M = 0$). Then we can deactivate subsumed clauses:

$$\begin{aligned}&\texttt{inductive}\ \mathsf{cdcl\_subsumed} :: \textit{'prag\_ st} \Rightarrow \textit{'prag\_ st} \Rightarrow \textit{bool}\ \texttt{where}\\ &\quad C \subseteq D \Longrightarrow \mathsf{count\_decided}\, M = 0 \Longrightarrow\\ &\quad \mathsf{cdcl\_subsumed}\ (M, N, U \wedge C \wedge D, \mathsf{None}, N_d, U_d)\\ &\qquad\qquad (M, N, U \wedge C, \mathsf{None}, N_d, D \wedge U_d)\end{aligned}$$

We combine these rules to express subsumption-resolution: We first learn the clause obtained by resolution. Then we can remove the antecedents. If either antecedent is in N, we also have promoted the conclusion from N to U. The advantage of our approach is that we can express other inprocessing techniques without adding new rules, only by specializing them.

Overall we have 9 rules with some overlap with CDCL (propagation and conflict), but mostly simplification of clauses (removing true clauses and false literals from clauses) and pure literal deletion: When a literal always appears positively (or always negatively), we can set this literal to be true unconditionally (later removing all clauses containing it): every model after adding the clause is also a model of the original set of clauses but not the opposite. This is the first transformation that does not preserve models in IsaSAT or any other verified SAT solvers.

Refinement of Subsumption-Resolution. While the definition of subsumption resolution is very simple, the refinement to code was challenging.

We verified forward subsumption [7] following CaDiCaL [5] (unbounded however, so all clauses selected heuristically are checked). We sort clauses by size and check if the current candidate is subsumed by one of the smaller clauses. Because we use two-watched literals, we need to distinguish between the binary clauses (than can produce new units) and the other clauses. At the end, we implemented two forward subsumption passes: one for binary clauses only and the other for larger clauses.

To subsume the candidates, we build occurrence lists and populate them with binary clauses, whereas Kissat [5] reuses watch lists. Moreover, for efficiency, we need a new marking data structure for efficient detection of subsuming-resolution.

4 Correctness of the Code and Completeness

Our specification model_if_satisfiable takes the multiset of clauses and returns a model (if there is one) or None if the clauses are unsatisfiable. Our implementation $\mathsf{IsaSAT}_{\mathsf{SML}}$ opts takes an array containing the clauses and returns an optional array containing the assignment, assuming that the clauses do not contain duplicated literals or the empty clause (precondition proper_lits_no_dups_$\bot$). The additional argument opts activates and deactivates certain techniques for solving. The following theorem states that our implementation refines the specification:

Theorem 1 (SML End-to-End Correctness) *The following refinement relation holds:*

$$\begin{aligned}&(\mathsf{IsaSAT}_{\mathsf{SML}}\ \mathsf{opts},\ \mathsf{model\_if\_satisfiable})\\ \in\ &[\mathsf{proper\_lits\_no\_dups\_}\bot]\,\mathsf{clauses\_assn}\to\mathsf{option\_model\_assn}\end{aligned}$$

The LLVM version is nearly the same. It can handle duplicated literals and the empty clause. Moreover, the new specification model_if_satisfiable_bounded allows for an *unknown* result if arrays would grow larger than the size permitted by the fixed bit-width. While this limit does not exist in Imperative_HOL, it exists in practice as no machine supports arrays that large. Therefore, we technically weakened our theorem, but did not change practical guarantees on the generated code. For $\mathsf{IsaSAT}_{\mathsf{SML}}$ we start [12] with 64-bit unsigned integers and only switch to GMP integers if the arrays grow too large.

Theorem 2 (LLVM End-to-End Correctness) *The following refinement relation holds:*

$$\begin{aligned}&(\mathsf{IsaSAT}_{\mathsf{LLVM}}\ \mathsf{opts},\ \mathsf{RETURN}\circ\mathsf{model\_if\_satisfiable\_bounded})\\ \in\ &[\mathsf{proper\_lits}]\,\mathsf{clauses\_assn}\to\mathsf{option\_model\_assn}\end{aligned}$$

Moreover, the change from SML to LLVM reduces the trusted code base: The Isabelle/LLVM model is closer to the actual LLVM, such that the trusted

pretty-printer is simpler. LLVM is also more low-level, such that fewer parts of the compiler have to be trusted. Finally, the LLVM compiler is more widely used and tested than the rather niche MLton compiler we used before.

5 Experience Porting the Development to LLVM

We report on the challenges we faced when updating the huge IsaSAT formalization (Sect. 5.1). Moreover, we report on the unverified parts of IsaSAT (Sect. 5.2), and finally compile some lessons learned (Sect. 5.3).

5.1 Required Changes

Before porting the development to LLVM, we removed our only remaining source of unbounded integers: the clause indices during the garbage collection. As garbage collection does not happen very often, we did not expect this to make a difference. Surprisingly, it turns out to have a performance impact.

Isabelle/LLVM is an entire tool set, including a fork of the original Sepref tool. While related to the original Sepref tool, there are different libraries, and the development of the two versions has diverged.

Initially, we tried to support both versions of Sepref. We ended up with two sets of files for code synthesis, and duplication of some libraries (to provide constants defined in Isabelle/LLVM but not in $\text{Sepref}_{\text{SML}}$). This significantly complicated our refinement approach, although we made it conceptually cleaner during the porting. Then, we realized that $\text{IsaSAT}_{\text{LLVM}}$ was much faster than $\text{IsaSAT}_{\text{SML}}$ (we observed a factor 2 on our test files), and decided to discontinue the SML backend.

With this, also some workarounds for SML specific performance issues (like the tuple `uint32 * bool * uint64` being much less efficient than combining the uint32 and the Boolean into a single 64-bit number) became obsolete.

Compilation. We have experimented with compilation flags before to improve performance. We know from the SML code that we need to increase the level of inlining, because many small functions make the verification easier. The same applies for LLVM and the easiest solution is to use link-time optimization that increases the inlining level as a side effect. However, this makes profiling impossible – exactly like the SML code. So there is no regression here.

Tuples. In 2021, we observed a major performance regression of the synthesis, caused by a new feature in $\text{Sepref}_{\text{LLVM}}$: pointer-equality tracking caused quadratic behaviour for case-splits of tuples. As our solver state is a large tuple, synthesis became impossible (several dozen minutes for simple functions).

To avoid the issue, we decided to work around on the abstract level, using getter and setter functions for the state's components, rather than case splitting. Now, every function on the state would first get the required components, update them, and then put them back. For example:

```
definition rescore_conflict :: clause_ index ⇒ isasat ⇒ isasat where
  rescore_conflict C S = do{
    let (M, S) = extract_trail S;
    ... (*reads the trail M and can change it*) ...
    let S = update_trail M S;
    RETURN S
  }
```

This makes synthesis much faster. However, the ownership model of Sepref does not allow aliasing, nor do our refinement relations allow leaving a 'gap' in the state where we moved out an element. As an easy work-around, we resorted to placing dummy-values, like empty lists, in the state, hoping that LLVM would optimize away the allocations and deallocations for these values. However, this did not happen: In the hot-spot of the SAT solver, the propagation loop, the dummy value for the trail was recreated and freed each time. Thus, we locally resorted to unfolding our code to make sure that we need only one free in the inner propagation loop. We leave a more principled solution of this problem (possibly changing Sepref) to future work.

We even attempted to go one step further (as the state-of-the-art SAT solver Kissat [4] does) and simply passing a pointer to the state structure as argument. Once we had already changed our refinement with accessors, we simply had to change them to work on a pointer. However, we never managed to make the synthesized code efficient. We observed a factor of 10 slower code. Hand-optimizing the accessors (basically making sure that LLVM understands that we care only about one component) reduced this to factor 2 slower. Once we realized that the LLVM optimizer was replacing the pointer by the structure passed directly as argument, we gave up on that approach.

5.2 Unverified Parts

In the generated SAT solver, there are some parts that we cannot verify. First, the parser is not verified, because the file system has no model in Isabelle (unlike CakeML, where conditions apply however). To this end, we link the verified code with an unverified C program, which provides the parser and command line interface.

Second, Isabelle/LLVM does not support any output (like statistics, or the DRAT proofs [28] required for the SAT Competition). For the SML version, we could use a feature of Isabelle's code generator to (axiomatically) implement a function by some external function (e.g. a function that does nothing in the model, by a printing function). As Isabelle/LLVM does not yet have such a feature, we resorted to post-processing the generated code (i.e., a function that does nothing in the model, is replaced by a printing function or even a function storing some literals for DRAT proofs). Note that this post-processing is not required for IsaSAT to work (but it won't print DRAT proofs).

5.3 Lessons Learned

Lesson 1: Embrace Duplication. We have already highlighted the importance of the set of all possible literals $\mathcal{A}$, in particular to establish a bound on the size of various arrays. At first, we tried to avoid duplicating this set across the different components on the specification side. This, however, resulted in a closer coupling of the various refinement proofs, impeding modularity: data structures that, conceptually, are just a small part of the whole state, have to be formalized on the whole state, just to have the set $\mathcal{A}$ available. We solved this problem by duplicating the set $\mathcal{A}$ on the abstract level for all new data structures. Note that this duplication is removed in a later refinement stage.

Lesson 2: The Limits are Isabelle Files. Checking our Isabelle files takes nearly two hours. This can be explained by three factors: 1. the heuristic and code synthesis amounts to 91 000 loc, making it a very large formalization; 2. the synthesis is single-threaded (for technical reasons); 3. Sepref encourages a style that is not very parallel: every refinement starts with a call to a tactic `refine_vcg` that generates the goals (meaning that all successive tactics have to wait). To improve performance we have attempted [12] to generate more standard proofs in Isar (by generating the text corresponding to the theorems to prove), but it is not clear that this style is faster as huge number of variables are generated (this style is required for more complicated proofs, however).

In order to improve Isabelle's performance and speed-up the testing of new heuristics in $\mathsf{IsaSAT}_{\mathsf{LLVM}}$, we have split the files into three parts: the shared definitions of the functions to refine, the (single-threaded) synthesis, and the correctness proof of the refinement. Even with these optimizations, proof checking still takes 2 hours. There is also no clear improvement path. The old SML version has a similar problem, but it is overall faster because it has fewer features, making it less critical.

Lesson 3: Performance Bugs exist. In order to improve performance, we need to measure and observe performance. To solve that problem, IsaSAT prints statistics and produces some timing information. The statistics during the run made identifying scheduling bugs for the different techniques possible – we accidentally ran some techniques way too often or barely ever. Especially because we increase the interval between two inprocessing rounds geometrically, a simple statistics at the end of the run is not sufficient. One interesting performance bug we found was that we accidentally inverted reducing clauses (marking them as removed) and garbage collection (physically removing them). Therefore, we would nearly always physically delete clauses. We never saw this issue, because we also printed the statistics inverted. To help debugging performance, we produce some timing information by measuring time in the C program:

```
c propagate           : 83.48% (581.66 s)
c reduce              : 0.12% (0.82 s)
c subsumption         : 0.06% (0.39 s)
c pure_lits           : 0.05% (0.33 s)
```

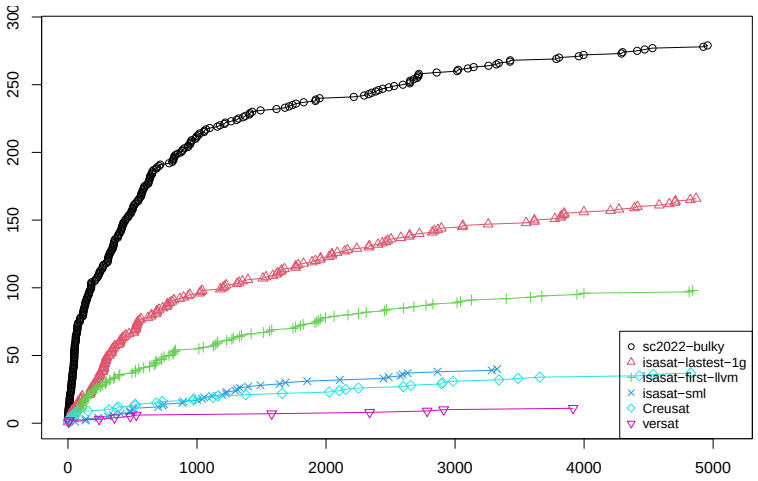


Fig. 1. CDF of the performance of SAT solvers

```
c binary_simp          : 0.02% (0.15 s)
c GC                   : 0.16% (1.10 s)
```

This helps to identify bottlenecks but also outliers where one technique is particularly slow and requires some limits or a change in the scheduling to avoid slowing down the solver too much. This makes it possible to identify errors like allocations in loops. The overall timing matches what we expect from other SAT solvers (although usually they spend more time on inprocessing and less on propagation).

6 Performance

In order to study the performance we have run 3 different IsaSAT versions: the original SML solver (using MLton with the LLVM backend), the first port of the IsaSAT solver, and the current version with inprocessing and various other improvements on heuristics that do not require any change on our PCDCL calculus, notably rephasing and target phases [10] (but no local search) and the alternation between aggressive restarts (heuristically seems better for UNSAT) and few restarts (seems better for SAT) following the ideas of Chanseok Oh [24].

We run all the benchmarks from the SAT Competition 2022 on an Intel Xeon E5-2620 v4 CPU at 2.10 GHz (with turbo-mode disabled) with a memory limit of 7 GB and a timeout of 5000 s. For comparison, we have included VERSAT [23] and CreuSAT [25]. For completeness, we have included Kissat [6] (more precisely the bulky version submitted for the anniversary track).

The results are given in Fig. 1 as a CDF (the higher the curve, the more solved problems). The first surprise is that CreuSAT performs similarly to $\text{IsaSAT}_{\text{SML}}$ (37 vs 40 solved problems), worse than expected given the results reported in the

Master's thesis [25] that tested on the 2015 benchmarks. We suspect that is due to the garbage collection and the fact that problems from the SAT Competition have become harder.

There is a clear improvement when going from the SML version to the LLVM version (98 solved), while the latest version solves 166. The SML version produces 335 out-of-memory errors (OOMs), the base LLVM version is more memory efficient (23 OOMs) like the latest IsaSAT version (19 OOMs) or CaDiCaL that has the same memory layout (17 OOMs). However, there is still a large gap to reach the performance level of Kissat and its inprocessing techniques.

7 Conclusion

We have reported on updating our verified SAT solver IsaSAT to a more powerful base calculus (our pragmatic CDCL) which can express inprocessing, and to the more efficient Isabelle/LLVM backend. We have also compiled important lessons learned from proof-engineering and maintaining large formalizations like IsaSAT (∼200 kloc of proofs).

Our changes made IsaSAT solve 4 times more problems (166/40), making it the most efficient verified SAT solver. At the same time, our verification is more complete than the next fastest verified solvers.

Most techniques (including the two most important, vivification and probing) either fit into our new PCDCL base calculus or do not require any change (like random walk [10] that is conjectured to be the reason for the major performance improvement in 2020). One major technique that we cannot currently express is variable elimination, because models are changed and need to be fixed. We leave the required extensions to our PCDCL for future work.

Acknowledgments The work presented here was done over several years and several work places. The first author was supported for some time by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), and the LIT AI Lab funded by the State of Upper Austria. We thank the anonymous reviewers for their detailed comments.

References

[1] Andrici, C.C., Ștefan Ciobâcă: A verified implementation of the DPLL algorithm in Dafny. Mathematics **10**(13), 1–26 (June 2022), `https://ideas.repec.org/a/gam/jmathe/v10y2022i13p2264-d850381.html`

[2] Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, Department of Computer Science, University of Helsinki, Finland (2022)

[3] Becker, H., Blanchette, J.C., Fleury, M., From, A.H., Jensen, A.B., Lammich, P., Larsen, J.B., Michaelis, J., Nipkow, T., Popescu, A., Schlichtkrull, A., Tourret, S., Traytel, D., Villadsen, J.: IsaFoL: Isabelle Formalization of Logic, `https://bitbucket.org/isafol/isafol/`

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

[5] Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2021. In: Proc. of SAT Competition 2021 – Solver and Benchmark Descriptions (2021), submitted.

[6] Biere, A., Fleury, M.: Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In: Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2022-1, pp. 10–11. University of Helsinki (2022)

[7] Biere, A., Järvisalo, M., Kiesl, B.: Preprocessing in SAT solving. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 391 – 435. IOS Press, 2nd edition edn. (2021). https://doi.org/10.3233/FAIA200992

[8] Blanchette, J.C., Fleury, M., Lammich, P., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. J. Autom. Reason. **61**(1-4), 333–365 (2018). https://doi.org/10.1007/s10817-018-9455-7

[9] Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings. LNCS, vol. 5170, pp. 134–149. Springer (2008). https://doi.org/10.1007/978-3-540-71067-7_14

[10] Cai, S., Zhang, X., Fleury, M., Biere, A.: Better decision heuristics in CDCL through local search and target phases. J. Artif. Intell. Res. **74**, 1515–1563 (2022). https://doi.org/10.1613/jair.1.13666

[11] Fazekas, K., Biere, A., Scholl, C.: Incremental inprocessing in SAT solving. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. LNCS, vol. 11628, pp. 136–154. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_9

[12] Fleury, M.: Optimizing a verified SAT solver. In: Badger, J.M., Rozier, K.Y. (eds.) NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings. LNCS, vol. 11460, pp. 148–165. Springer (2019). https://doi.org/10.1007/978-3-030-20652-9_10

[13] Fleury, M.: IsaSAT and Kissat entering the EDA Challenge 2021 (2021), https://www.eda-ai.org/results/, system description accepted at the EDA Challenge 2021. Available at https://m-fleury.github.io/ox-hugo/Fleury-EDA-Challenge-2021.pdf

[14] Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings. LNCS, vol. 6009, pp. 103–117. Springer (2010). https://doi.org/10.1007/978-3-642-12251-4_9, https://doi.org/10.1007/978-3-642-12251-4_9

[15] Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning - 6th International Joint Conference,

IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings. LNCS, vol. 7364, pp. 355–370. Springer (2012). https://doi.org/10.1007/978-3-642-31365-3_28, https://doi.org/10.1007/978-3-642-31365-3_28

[16] Lammich, P.: Automatic data refinement. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 84–99. Springer (2013). https://doi.org/10.1007/978-3-642-39634-2_9

[17] Lammich, P.: Refinement to Imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 253–269. Springer (2015). https://doi.org/10.1007/s10817-017-9437-1

[18] Lammich, P.: Generating verified LLVM from Isabelle/HOL. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA. LIPIcs, vol. 141, pp. 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.22

[19] Lammich, P.: Efficient verified (UN)SAT certificate checking. J. Autom. Reason. **64**(3), 513–532 (2020). https://doi.org/10.1007/s10817-019-09525-z

[20] Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. pp. 75–88. IEEE (2004). https://doi.org/10.1109/cgo.2004.1281665

[21] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001. pp. 530–535. ACM (2001). https://doi.org/10.1145/378239.379017

[22] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)

[23] Oe, D., Stump, A., Oliver, C., Clancy, K.: versat: A verified modern SAT solver. In: Kuncak, V., Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation, LNCS, vol. 7148, pp. 363–378. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-27940-9_24

[24] Oh, C.: 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. pp. 307–323 (2015). https://doi.org/10.1007/978-3-319-24318-4_23

[25] Skotåm, S.H.: CreuSAT – Using Rust and Creusot to create the world's fastest deductively verified SAT solver. Master's thesis, University of Oslo (2022), https://www.duo.uio.no/handle/10852/96757

[26] Tan, Y.K., Heule, M.J.H., Myreen, M.O.: cake_lpr: Verified propagation redundancy checking in cakeml. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. LNCS, vol. 12652, pp. 223–241. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_12

[27] Weeks, S.: Whole-program compilation in MLton. In: Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006. p. 1. ACM Press (2006). https://doi.org/10.1145/1159876.1159877

[28] Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer (2014)

9 Beyond Verification: Certification

Workshop Paper

[1] Fleury, Mathias and Armin Biere. *Scalable Proof Producing Multi-Threaded SAT Solving with Gimsatul through Sharing instead of Copying Clauses*. Ed. by Matti Järvisalo and Daniel Le Berre. I helped in the design and wrote parts of the paper. 2022. URL: https://arxiv.org/abs/2207.13577.

Scalable Proof-Producing Multi-Threaded SAT Solving with Gimsatul through Sharing instead of Copying Clauses

Mathias Fleury and Armin Biere

Albert-Ludwig-Universität Freiburg, Germany
fleury@cs.uni-freiburg.de biere@cs.uni-freiburg.de

Abstract

We give a first account of our new parallel SAT solver Gimsatul. Its key feature is to share clauses physically in memory instead of copying them, which is the method of other state-of-the-art multi-threaded SAT solvers to exchange clauses logically. Our approach keeps information about which literals are watched in a clause local to a solving thread but shares the actual immutable literals of a clause globally among all solving threads. This design gives quite remarkable parallel scalability, allows aggressive clause sharing while keeping memory usage low and produces more compact proofs.

1 Introduction

The SAT Competition is the place to show off the newest and fastest SAT solvers for many years now. To improve reliability and increase correctness, every solver in the main track must produce a DRAT certificate that is checked by the official checker DRAT-TRIM – and only checked problems count as solved. Solvers implementing techniques that cannot be expressed in DRAT can only take part in the no-limit track (even if no technique seems to bring an edge over other main-track solvers). While sequential solvers have improved considerably recently, making use of multiple CPU cores can improve performance even further. Therefore the SAT Competition has a parallel and a cloud track. In those track no proofs are required.

There are several ways to parallelize SAT solvers (Section 2), but most solvers in both the parallel and the cloud track rely on the common portfolio approach pioneered by MANYSAT [7] enhanced by exchanging clauses. The idea is to rely on existing fast single-core SAT solvers without the need to modify them beyond possibly adding a mechanism for exchanging clauses. The main approach to exchange clauses between solver threads is copying them.

The main argument in favor of this approach is that adding a light-weight synchronization to existing code is simple and does not need to change the sophisticated and complex core data-structures of the solvers. We argue that this argument leads to sub-optimal results both with respect to scalability and memory usage. Instead we propose to physically share clause data (through pointers) during clause exchanges instead of copying them.

More precisely, the classical portfolio approach generates two issues (Section 3). First, it increases the amount of memory needed by each solver thread. Second, even if proof production was implemented, it produces larger proofs than necessary by requiring duplicating shared clauses in the proof. By actually sharing instead of copying clauses we can address both issues.

Current proof checkers do not support exchanging clauses between certificates and it is not obvious how to do this correctly, because clause exchanges go in both directions. The other issue is that proof checking cannot completely independently be done in parallel, again, because the solver threads exchange clauses. One partial solution to the problem is to log all the clauses into a single file and have a global clock to serialize all the derived clauses in order. However this solution still requires to duplicate exchanged clauses in the proof log, leading to large traces and long checking times, as also our experiments confirm.

In order to reduce this work, we have started a new SAT solver called GIMSATUL that aggressively shares clauses among the solver instances. We revisit an old idea that co-existed with the currently dominating exchange-only approach until around 2011, which on the other hand pre-dates more sophisticated proof tracing techniques used in the SAT competition now as well as all the improvements to sequential solving made in the last decade. We observe that physically sharing clauses not only makes it possible to reduce the memory footprint, but also enables sharing in the proof log, thus reducing the amount of work required for checking. While these changes have a large impact on the core data-structures of the SAT solver they do not require any change to existing proof checkers.

We implemented our new solver GIMSATUL in 13 kloc of C. It is available online[1] and uses atomic operations to adjust reference counters and exchange pointers, as standardized with C11 in `stdatomic.h`. It relies on the Pthreads programming model for threads, locking, and condition variables. Furthermore it uses lockless fast-path code whenever reasonable.

The name GIMSATUL is derived from *gimbatul* in the "Black Speech" language invented by R. Tolkien and occurs in the inscription of the "One Ring" in "Lord of the Rings" and literally translates to "find them" (all). We follow that terminology in the paper and in the source code. Accordingly the main thread which performs preprocessing sequentially and organizes everything is called the *Ruler* and an actual solver thread is called *ring*.

This paper is a slightly extended version of our POS'22 paper made available to attendees of the workshop on Pragmatics of SAT (POS'22). In this paper we use additional space to improve readability of figures by increasing their size. We further include various comments present in the original longer submission, which had to be removed due to the page limit for the final version at the workshop of 14 pages (plus references).

Besides briefly going over the architecture of the solver we focus in this paper on describing differences to the core data-structures compared to other solvers. In particular watched literals cannot be kept as the first two literals in the clauses, since different solver instances may need to watch different literals (Section 4). The solver also performs sequential inprocessing, requiring to give back all irredundant clauses to a single instance. This new infrastructure makes sharing possible and space efficient. The main advantage as highlighted by our experiments is that our solver scales *linearly* with the number of threads. Another major consequence is that sharing ensures that less memory is required to run GIMSATUL (Section 5).

Finally, we compare DRUP/DRAT proofs generated by GIMSATUL (Section 6) with the version of the solver where sharing is not done at the proof level and instead clauses are duplicated in the proofs (Section 7). We show that proof size is largely independent of the number of threads.

2 Parallel SAT Solving and Related Work

This work does not attempt to define what SAT is and how SAT solvers works in details. For those details, we refer to the *Handbook of Satisfiability* [4]. Detailed knowledge about the inner working of SAT solvers is not required beyond the fact that SAT solvers resolves clauses to derive new clauses and that those clauses can be exchanged if two solvers work on the same problem. Non-satisfiability preserving transformation are not done in parallel.

With respect to parallelization, we classify the approaches used by SAT solvers into three different categories, as shown in the following table:

[1] https://github.com/arminbiere/gimsatul

	One solver		Several solvers	
One Problem	CDCL + simplification (e.g., KISSAT [3])	(1)	Portfolio (e.g., MALLOB [16])	(2)
Multiple Problems	Cube-and-conquer (CnC) by hand (e.g., Marijn Heule [9])	(3)	CnC + resplitting + sharing (e.g., PARACOOBA [3])	(4)

Note that some solvers also combine techniques like PAINLESS [6] from (2) and (4), but in their default configuration from the SAT Competition, they use Approach (2). In this work we reconsider Approach (2) where current solvers rely on mostly unmodified single-core SAT solver engines. The current state-of-the-art solvers rely on (logically) *exchanging* clauses through copying but not physically sharing them.

Usually this approach further takes advantage of the portfolio idea: different solver threads use different strategies, e.g., different restart scheduling, decision heuristics, etc. The hope is that different instances learn clauses useful in other threads, especially short clauses.

GIMSATUL is far from being the first solver to use physical clause sharing. One solver, SARTAGNANG [11] was discussed at the Pragmatics of SAT workshop in 2011. Unfortunately, no performance discussion was done to see how the solvers scales per thread. However, this solver was not faster that clause exchanges. A more interesting difference is that they use one thread to simplify the problem. Therefore, they have adapted the messages that are exchanged: Instead of only exchange clauses, the message can also be that the clause is subsuming another one which can be removed. An interesting observation is that they save where the watch list was found last in one of their configuration, which is similar to caching during search.

Other solvers like PAMIRAXT [17] use a combination of cube-and-conquer and portfolio: The search space is initially divided and each space is solved using several threads sharing clauses. On motivation for the space splitting is that their implementation shares all clauses (among the instance working on the same sub-problem). This is too much for the poor SAT solver instances especially when 32 threads learn clauses at the same time.

For detailed information on the architecture or the solver that used physical sharing of clauses before 2011 (even though none of these solvers seems to be maintained anymore) we refer to the corresponding chapter in the *Handbook of Parallel Constraint Reasoning* [1]. Most current research tries to improve the portfolio approach and investigates better selection of clauses to exchange. Another interesting idea is to let a GPU select the clauses which are useful [14], partially based on the idea the clauses that would have produced a conflict earlier are likely useful in the future too. We focus on scalability and proofs and leave this aspect to future work.

3 Proof Checking

In the previous section we discussed the different approach to solve problems. Proof checking has a different flavor:

	One solver		Several solvers	
One Problem	one checker (e.g., DRAT-TRIM [21])	(1)	previous work and this paper (e.g., DRAT-TRIM [21])	(2)
Multiple Problems	parallel checker (e.g., CAKE_LPR [19])	(3)	none	

Checking (1) is the most standard and best understood approach, even if there is some technical issue on the semantics of (reused) units [15]. The approach (3) is very promising to check

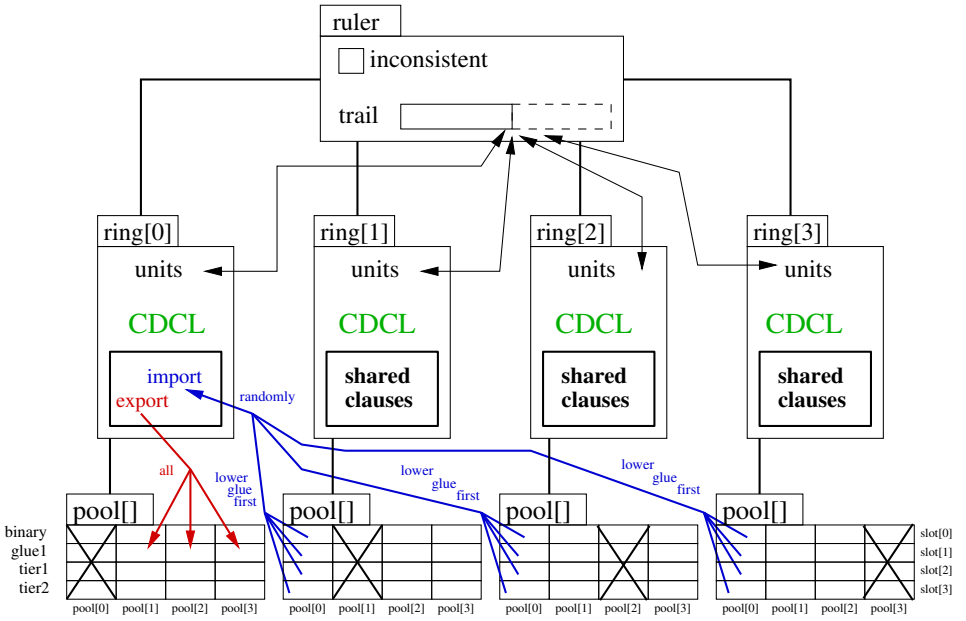


Figure 1: Exporting and importing learned clauses, including the empty clause through the "inconsistency" flag, learned units, and low glue clauses. Irredundant (including original) clauses are shared up-front while cloning the initial clause data-base (after pre- and inprocessing).

cube-and-conquer proofs. The checker checks each proof (i.e., the one with the cubes) and checks that the cubes cover the entire search space. All those checks can be done in parallel. There are some limitations in the verified proof checker of (3); for example, cubes generated by March cannot be used because trivially unsatisfiable cubes are removed from the clauses, and more critically, the "exchange" of information that the cubes are unsatisfiable is done via (forgeable) command line arguments (while the checking itself is verified in CakeML).

In this paper we attempt to both improve the memory requirement by sharing physically clauses and also, as a side effect, to reduce the size of the proofs.

4 Gimsatul

The key difference of our new solver Gimsatul compared to all recent portfolio solver is the physical sharing of clauses. In order to do so, we have to revisit the implementation of watched literals, a data structure to identify propagation and conflicts (Section 4.2). Invariants on watched literals also limit how clauses can be imported by an instance (Section 4.3). We further discuss how model reconstruction is used to handle units (Section 4.4) and how sequential inprocessing can be achieved with clause sharing (Section 4.5).

4.1 Overall Organization

The core SAT solvers in Gimsatul follow the design of our other recent solvers, particularly they share many ideas with the "sc2022-light" version of Kissat. For instance it includes rephasing [2] and the stable and focused mode. There are two differences worth mentioning. First, the SAT solver does not use an arena to compactly represent the clauses in memory. One motivation for this trick is to put clauses consecutively in memory in the order in which the solver accesses them [18]. It further allows efficient garbage collection during clause data-base reduction (forgetting heuristically-unimportant learned clauses). This is something that we cannot do for Gimsatul because each ring (solver instance) completely independently allocates and reduces clauses in memory. Hence, we allocate memory with `malloc`. Second, we use two watch lists, one for irredundant binary watch lists (shared across the instances) and one for watching clauses, unlike the single watch list used in our other solvers.

Overall the solving process looks as follows. First, the main thread called *Ruler* parses and allocates all clauses. At that point, the *Ruler* owns all the clauses. It then enters a "cloning" phase, which first passes all its clauses to the first ring as new owner. It also creates the shared global watch lists for irredundant binary clauses. This first instance is then forked into as many instances as needed, to match the number of requested solving threads given on the command line, sharing all large clauses through new sets of watched literal lists. However, each solver instance reuses the same watch lists (a flat literal array) for the irredundant binary clauses.

After cloning, solvers start solving the CNF individually, importing and exporting learned clauses (Section 4.3). The first solver instance to determine the problem as solved is declared the winner and a termination flag is raised, forcing other solver threads to exit their CDCL loop too. If the winner deduced the empty clause the problem is unsatisfiable. Otherwise it has found a satisfying assignment which is then extended by the main thread to a full model using the reconstruction stack of the *Ruler*/*Simplifier* produced during pre- and inprocessing.

All solver instances logically reclaim (dereference) clauses satisfied by learned root-level units, as part of frequent clause-database reductions, which mainly have the purpose of discarding useless learned clauses. However, reclaiming in this context just means decreasing the reference count of a clause. Only until the satisfying root-level unit has reached all solver instances and is picked up during clause-database reduction, that root-level clause is finally physically deallocated (and is also deleted in the proof trace).

Cleaning the clauses from root-level falsified literals is much more involved, as the actual clauses can not be shrunk in parallel – the blocking and watched literals potentially have to be changed. For that purpose the first ring is responsible for starting simplification rounds during which all irredundant clauses are handed over back to the *Ruler*. Redundant watches pointing to redundant clauses are saved locally for each ring separately. After this preparation phase (called "uncloning" in the code), the inprocessing can start.

As clauses are shared we do not remove falsified literals during solving. Instead solvers synchronize regularly and *hand* back their clauses to the initial thread which then, in the role of the *Ruler* process, becomes responsible for (*i*) removing satisfied clauses (which is something that can already be done by each ring by marking the clause as removed locally), (*ii*) removing false literals from clauses, and (*iii*) renumbering literals if holes appeared to keep literals compact. After that, the heuristics are adapted (by renaming literals in those heuristics too).

Besides sharing clauses and waiting for each other for inprocessing, all solver instances are independent of each other and can run using a different strategy. We currently use a simple and limited portfolio: we use a different initialization for the first random walk (limited local search) that initializes the phases, which is one of the phases of our rephasing strategy [2]. The result of this walk is exported and used as saved phases initially, leading the solver to different

Ring number mod 12	0	1	2	3	4	5	6	7	8	9	10	11
Mode	S+F	S	F	S+F	S	F	S+F	S	F	S+F	S	F
Phase	0	1	0	1	0	1	0	1	0	1	0	1
Reason Bumping	0	0	1	1	0	0	1	1	0	0	1	1

Table 1: Strategy of each thread (S = stable, F = Focused)

search directions. Interestingly, we initially had no diversification at all and already observed improvement in the performance of the solver, due the exchange of clauses (described in more details in Section 4.3). In Table 1, we show the amount of diversification we do, but note, that we currently only have 12 different configurations.

4.2 Revisiting Propagations

Since zCHAFF [12], all modern SAT solvers use *watched literals* to identify clauses that can propagate or are in conflict. The idea is to distinguish two literals in a clause. Whenever either of those literals is set to false, then the clause must be checked to see if it can propagate information or is conflicting. Otherwise, the clauses do not need to be updated (nor visited). This is essential to make propagations and conflict detection efficient. Originally pointers to the two watched literals in each clause were used. Modern implementation however make sure that the first two literals of the clause are watched, by swapping clause literals during propagation.

However, insisting on watching exactly the first two literals does not work when sharing clauses because different solver threads (rings) have different partial assignments (and trails) and thus usually watch different clauses and literals in those clauses. Hence when sharing clauses we can not swap literals in the literal list of a clause anymore.

Our solution is to store the watched literal pair in a watcher data-structure separately, which also has a pointer to the immutable actual clause. Those watchers are *not* shared amongst solver instances, except for irredundant *binary clauses*, because they do not require any changes. It is important to notice that, as the clauses are not changed, there is no need for locks when accessing a clause by different threads.

The pair of literals in our thread-local watcher data-structure also serves as "blocking literals" which are checked first to satisfy the clause. This is a common technique to reduce the number of times the actual clause data has to be accessed. If the blocking literal check fails, however, this scheme incurs an additional pointer access compared to the standard version of placing the two watched literals in front of the literal list of a clause, effectively merging the clause and the watcher data-structure. On the other hand our watcher structure is much more compact than a full clause and thus likely has better cache locality.

4.3 Clause Sharing

An important design choice for parallel solvers is to determine which clauses to share and how often they are imported. We use a very simple policy: (*i*) at most one clause is imported before each decision and (*ii*) when learning a clause with a low glucose level, we immediately export it. We do however import and export all derived root-level assignments eagerly as well as check for termination and thus "inconsistency" (another thread proved the formula to be unsatisfiable). Figure 1 shows how the sharing happens and is described in the text below. All rings share the trail composed of unit literals.

Which Clauses to Import. When importing clauses before a decision, only a single one is imported at a time. To import and export clauses, each ring has 4 slots for each other ring with clauses to export: one for binary clauses (64 bits representing two 31-bit literals), glue 1 (non-binary) clauses, glue-2 clauses (remaining tier1 clauses), and finally tier2 clauses (glue 3 to 6). In Figure 1, the slots are called *pools* and the pool from a ring for itself is crossed out (because it is useless to have a ring share a clause with itself).

Before every decision, each ring attempts to import one clause after checking that no new units should be imported first. Without any new unit it selects its pool among the shared pools of another randomly chosen ring. The goal of randomly picking the exporting ring is to implement a global (bounded) queue with relaxed semantics [8], i.e., a bounded k-queue, which probabilistically guarantees low contention. Each queue is implicitly bounded because threads during exporting learned clauses simply overwrite references and thus drop clauses with the same glue class (binary, tier1, tier2, tier3) as the exported clause.

Then the importing ring checks its slots in order with lowest glue first (this is a fast-path without locking) if there is any clause to import (the blue arrows in Figure 1). If one is found, then the slot is emptied with an atomic exchange operation (thus requiring 64-bit word size).

Importing Clauses. As described in the previous paragraph, the solver imports one clause at a time. The current partial assignment on the trail has to be fixed if the clauses is propagating/conflicting. Otherwise, some literals (not arbitrary ones) have to be selected as watched literals to fulfill the watch list invariants.

Before we actually import a clause it is checked for not being already subsumed by another existing clauses, using the watch list as an approximation for occurrence lists. This forward subsumption check traverses the watcher list of a new watched literal which is smaller and thus might miss some subsuming clauses, but is complete for exact matches (identical clauses ignoring root-level falsified literals). Besides adapting the current interpretation to the single imported clause, importing simply means watching the correct literals and adding the clauses to the correct watch lists.

In our implementation importing a clause amounts to increasing the overall number of occurrences of that clause. We rely on atomic operations to adjust these counters (using `atomic_fetch_add` and `atomic_fetch_sub` from `stdatomic.h` in C11).

In contrast to operating in single-threaded mode and unlike all our single-threaded SAT solvers Gimsatul using multiple threads is highly *non-deterministic*, because importing clauses is done eagerly and depends on the exact thread scheduling, in which order memory accesses occur and caches are updated etc.

Exporting Clauses. Important learned clauses with small glucose level are exported immediately during the conflict analysis, with references to the exported clause added to the $n-1$ pools of the exporting thread (the red arrows starting from ring 0 in Figure 1). Each pool corresponds to exactly one thread and has four slots of clauses references sorted by glucose level (binary, tier1, tier2, tier3 clauses). This allows to share these important learned clauses with all other solver threads, prioritized by importing low-glue clauses first.

Life and Death of Clauses. Remark that the clauses are imported according to the score they had previously and are handled the same way as every other learned clause. Therefore, low LBD clauses are never removed (in particular binary clauses), but unused tier2 clauses will eventually be deleted. Large tier1 clauses might be removed by vivification though.

During execution, the LBD score of clauses can actually change. This update is for example in Glucose for clauses involved in the conflict analysis (and only if the score decreases), although the actual LBD of the score is already defined when the clause is propagating. An interesting question is whether *promoted* clauses, i.e., clauses whose LBD changes enough to

become tier1 or tier2 clauses should be shared. We experimented with promotion, but did not observe any benefit. It is also rather difficult to implement as it might either result in glue values to diverge between clauses and their watches or otherwise requires atomic update of glue values in clauses.

4.4 Model Reconstruction

GIMSATUL relies on the model reconstruction [10] to produce a model and "undo" the inprocessing. This reconstruction stack is not shared amongst the rings. Instead, a single one is used. This is sufficient because all solvers work on the same formula.

Unlike previous solvers, we actually go one step further and completely remove fixed literals assigned at root-level (decision-level zero). In KISSAT we would also remove those literals but not remove them from external partial assignment Therefore, we do not need to put those literals on the reconstruction stack. Reusing the reconstruction stack is in particular used for units derived during inprocessing techniques (see more details in the next paragraph), because it avoids communicating such literals back to each SAT instance.

4.5 Inprocessing

We have implemented two different kinds of inprocessing in GIMSATUL. Some transformations are part of probing like vivification. They run directly in the different rings but they do not shorten shared clauses. Instead a new shortened clause is added and the other is removed from the clause set. Other inprocessing techniques require to change the set of original or in general irredundant clauses such as bounded variable elimination [5]. For those transformation, each instance first gives up all references to the clauses. At the end, only the first instance knows the location in memory of all the clauses. It gives them back to the *Ruler* instance which then starts the *Simplifier*. This *Simplifier* is then in charge of transforming all the irredundant clauses (including shortening and strengthening them) using the standard algorithms, e.g., for variable elimination. After that the clause are passed back to the first ring which in turn passes them back to the other rings (with the shared watch list of binary clauses).

Getting this "uncloning " option to work was rather challenging, because units produced must still be shared amongst all rings. This in turn can enable more propagations at root-level (at decision-level zero) that again needs to be shared. If this is not done properly, some units might get lost, which is an issue as then the candidate model of the different instances might not know about those units, leading potentially to incorrect models: If the *Simplifier* removes irredundant clauses containing a given literal, because this literal is not assigned and can appear in the redundant clauses, it can get assigned to the opposite value leading to an incorrect model.

5 Experiments on Scalability

We ran experiments[2] on our cluster equipped with Intel Xeon E5-2620 v4 CPU at 2.10 GHz (with turbo-mode disabled) with a memory limit of 128 GB for each node. Those CPUs have 16 real cores and 32 cores using hyper-threading. In order to keep the testing time reasonable (and running solvers in parallel), we assigned 127 GB for 8 cores or more, 63 GB for 4 cores, 31 GB for 2 cores, and 15 GB for 1 core.

Performance. We first compare the overall wall-clock-time performance of our solver GIMSATUL to the state-of-the-art solver P-MCOMSPS [20], which won the parallel track of the SAT

[2]Experimental data available at https://cca.informatik.uni-freiburg.de/pos22gimsatul.

	solved	sat	uns	elapsed time (s)	PAR-2 (10^3)	space
GIMSATUL-32	310	**151**	159	3 928 732	1 031	1 578 816
P-MCOMSPS-32	**315**	141	**174**	5 804 244	1 037	3 038 965
GIMSATUL-16	308	149	159	2 069 523	1 055	843 444
P-MCOMSPS-16	309	136	173	2 607 129	1 076	1 495 201
GIMSATUL-64	304	150	154	4 768 809	1 119	2 948 359
GIMSATUL-8	298	144	154	1 223 199	1 178	460 013
P-MCOMSPS-8	297	131	166	1 584 009	1 229	723 740
GIMSATUL-4	282	138	144	744 566	1 371	263 227
P-MCOMSPS-4	260	119	141	854 403	1 614	317 691
GIMSATUL-2	262	130	132	499 420	1 633	167 771
GIMSATUL-1	230	117	113	263 851	1 963	80 947
P-MCOMSPS-2	204	87	117	679 142	2 187	187 676

Table 2: Results on the problems from the SAT Competition 2021

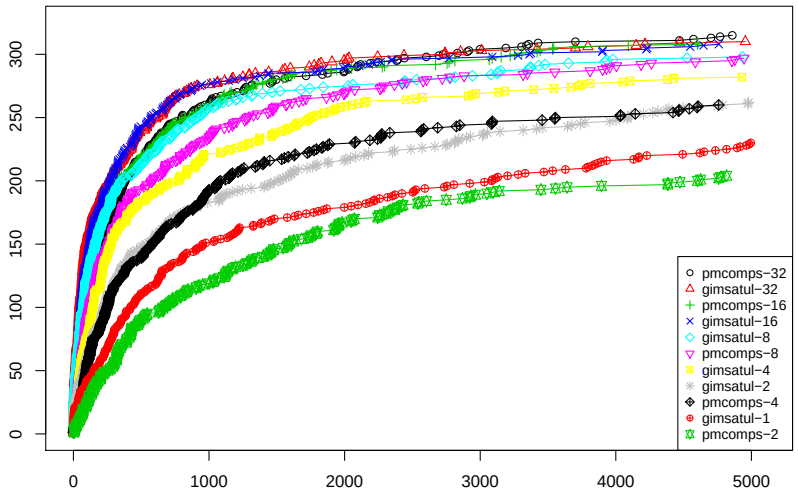


Figure 2: Cumulative distribution function (CDF) of the wall-clock solving time of both GIMSATUL and P-MCOMSPS for various considered number of threads.

Competition 2021. It features an advanced SAT core solver and incorporates many ideas to improve parallel solving including sophisticated diversification techniques. However, it fails when run with a single thread (producing an exception). We assume that this is due to the fact that one thread is used to strengthen clauses and the others for solving. Therefore we only consider experiments for P-MCOMSPS with at least two threads. The CDF (Figure 2) and the raw results (Table 2) give the results for this initial experiment.

At first we were surprised by the fact that P-MCOMSPS needs twice as many threads to match the performance of GIMSATUL for less than 8 threads. Due to time constraints, we could not run all the configurations on the SAT Competition 2020 too. But partial runs show that

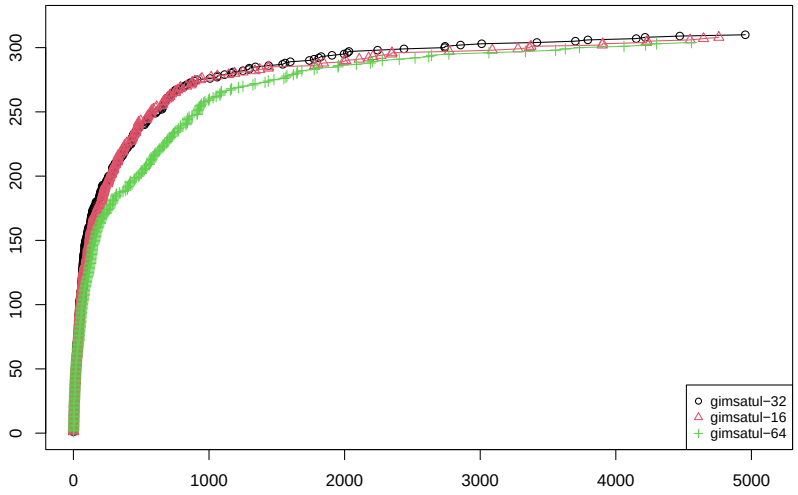


Figure 3: CDF with more threads than cores

the results are very similar for 2, 4, and 8 threads (i.e., the performance of P-Mcomsps with n threads is similar to the performance of Gimsatul with $n/2$ threads). This shows that our results do not seem to be biased to the 2021 benchmarks (a valid threat to validity).

The plot also shows that Gimsatul is in general faster than P-Mcomsps, particularly with respect to the PAR-2 score, even though both solve a similar number of benchmarks.

We have also experimented with higher number of threads than our machines support (Figure 3). The performance gap when using all 32 virtual threads on the 16 "real" cores does not yield a performance decrease. However, when using 64 threads, performance decreases. This indicates that Gimsatul does not spend all its time waiting for the other threads.

Scalability. While we only considered wall-clock-time performance above it is also interesting to investigate how effectively compute resources are used. This kind of question can arise in a cloud context where customers pay only for what is used and do not want to provision redundant compute resources (cores) unless solving latency is reduced effectively. We try to answer this question by plotting CDFs where (wall-clock) time is replaced by "time∗number of cores". This is equivalent to run our multi-threaded version on a single core.

For Gimsatul (Figure 4), we see once saturation is reached (i.e., after 32 threads on our 16 "real" core nodes), performance decreases. The 16-thread version has a disadvantage at the beginning, but catches up. One reason might be that our portfolio has only 12 different policies, and the 4 other rings run the same first four policies again. Even though the behavior is not deterministic, more diversification is probably better. It is significant to see that the other curves are basically identical. This shows that our solver scales with the number of cores. As future work we want to repeat this experiment with disabled diversification (portfolio).

For P-Mcomsps (Figure 5), the picture is different. Remark that the 2-thread version stops at 11 000 s, i.e., exactly the timeout of 5 500 s when you have 2 threads. First the performance loss for 32 threads is already much more pronounced. So virtual threading seems to be very harmful. Second, we can see that performance for 2 threads is worse than for the other configuration. We attribute this to a lack of optimization and diversification for this case. Third, peak performance

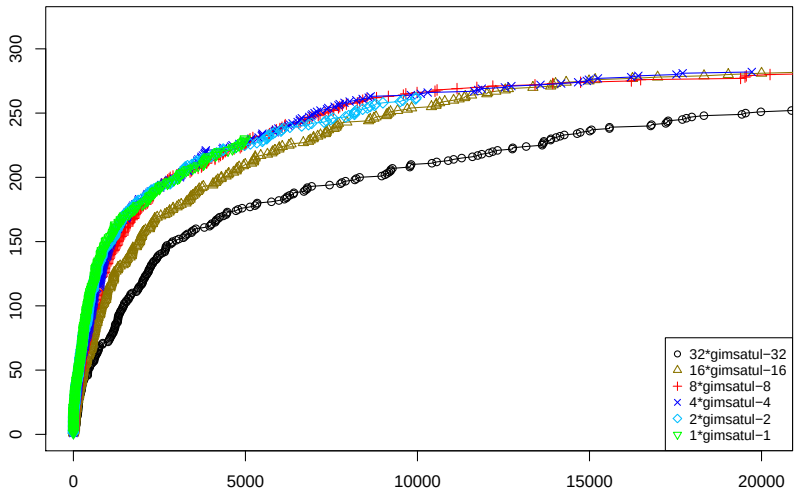


Figure 4: Scalability Gimsatul

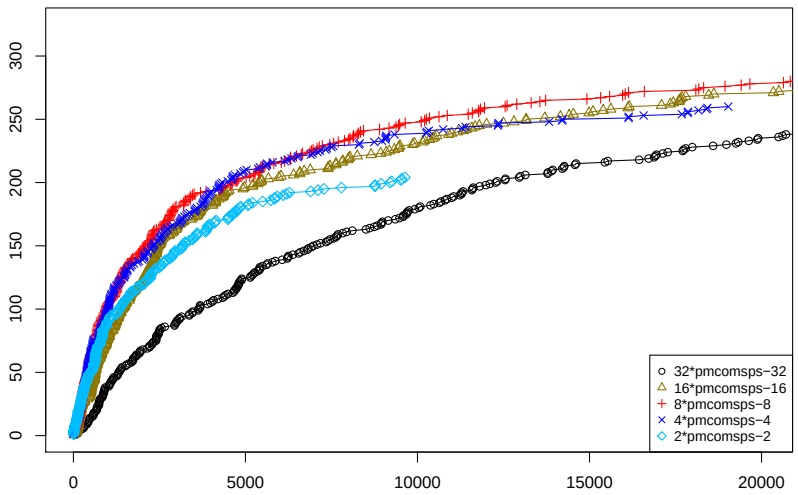
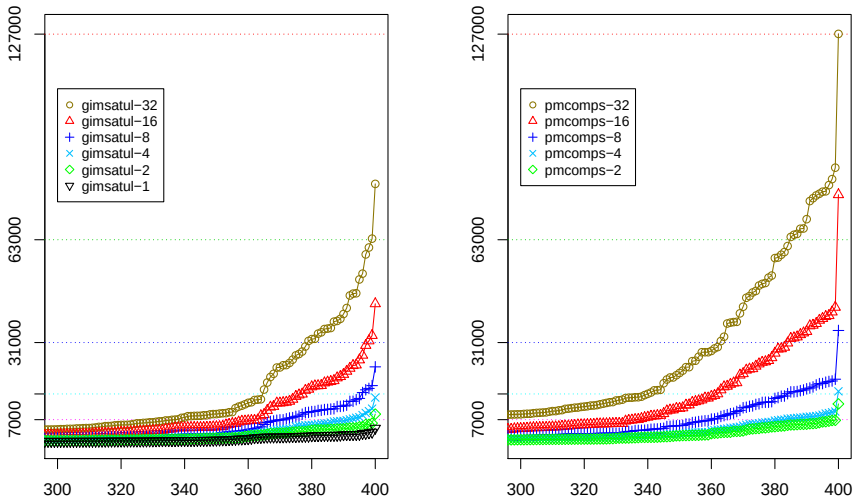


Figure 5: Scalability P-Mcomsps



(a) Memory usage of Gimsatul (b) Memory usage of P-Mcomsps

Figure 6: Memory usage in MB of both solvers on problems of the SAT Competition 2021

seems to be reached already for 8 threads, but the performance decrease compared to 16 threads is limited: it is more important to go from 4 to 8 threads, than to go from 8 to 16.

Memory Usage. In order to compare memory usage we have prepared two different plots. The first (Figure 6) shows peak memory usage (maximum resident-set-size) during the run. We can clearly see that Gimsatul uses much less memory than P-Mcomsps. For 32 threads Gimsatul only needs in one case for 32 threads slightly more than 80 GB and otherwise stays below 64 GB, which is half the memory available on our cluster nodes. On the other hand, also for 32 threads, P-Mcomsps hits the memory limit of 128 GB once.

In order to check how sharing works, we also checked the amount of memory used after the first preprocessing round but before cloning and any learning is done (Figure 7). This value is interesting because it shows an (optimistic) view on the cost of duplicating the watch lists and all other data-structures of the solver. With only binary clauses, the overhead would be very low. Without any binary clause all watch lists are duplicated, even though the actual clause data - the literals in clauses - are shared. Here we show how much more memory is used once Gimsatul has initialized all the different rings. We can see that the increase is on average much lower than the number of solver instances, i.e., the number of threads.

6 Generating DRAT Proofs

In essence, a DRAT proof certificate is a list of derived clauses ending with the empty clause (if the problem is deemed unsatisfiable). The key idea is that adding new derived clauses is satisfiability preserving: if the initial problem has a model, then the model with the new clauses has a model. Such clauses are said to be *redundant*. In general such certificates have a multiset semantics: the checker keeps the clause as many time as it was added (and removing one of the copies only removes one of them, not all). In Gimsatul, we know how often a clause is

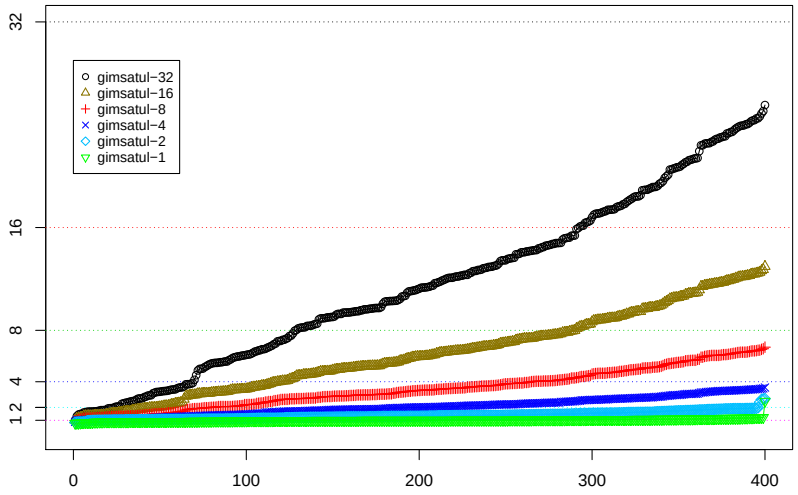


Figure 7: Initial relative memory increase of GIMSATUL in terms of resident size before cloning and after cloning, i.e., after preprocessing but before any solving/learning takes place

present (so we can keep only one clause) but this is not possible in general for parallel SAT solvers, requiring a copy for each shared clause.

The DRAT proof format was selected to be easy to implement for single threaded SAT solvers: It is sufficient to dump the derived clauses in order and to stop when the empty clause is derived. For parallel SAT solvers, it is tempting to produce one proof file per solver instance, but this does not work when the different instances exchange clauses. Instead we focus on a variant that produces a *single* proof file. This approach is not a new approach, but we did not find a description of it in related work.

One trivial solution is that every derived clause is written to the proof file in the order it is created and additionally added again (with multi-set semantics) when the clause is imported by another solver thread. This makes sure that the individual solver instances can delete individual clauses as they like and log these deletion steps independently of each other in the proof file.

With this trivial "copying" solution clauses are repeated as many times as they are copied, increasing the size of the proof file almost linearly with respect to the number of threads (core solver instances). We propose instead in this work to *share* those clauses in the proof instead of duplicating them, which in turn requires to share them among the solver threads too. Unfortunately, this requires some substantial changes to the data-structures for watched literals, as described in the next section, and is one of the main reasons we started GIMSATUL from scratch instead of incorporating similar ideas into our sequential state-of-the-art solver KISSAT.

Generating proofs is very easy in GIMSATUL: Every large (non-binary) clause has an atomically incremented and decremented reference counter for the number of occurrences in different solver threads (rings), occurrences in clause pools and during simplification phases in the simplifier. When creating (allocating) the clause, it is *added* to the proof. Sharing a clause consists of passing a pointer and incrementing (atomically) the reference count. Dereferencing a clause (for instance during clause-database reduction) decrements the reference count and when the reference counter reaches zero the clause is deallocated and at the same time marked

as *deleted* in the proof trace. In the meantime the clause is considered alive. As binary clauses are virtual and only occur in the watch lists, they are handled as in other solvers with virtual binary clauses as long as proof tracing is concerned.

All solver threads share the same proof trace file and writing to that file is synchronized implicitly using the standard locking mechanism of file I/O in `libc`. In particular, the library makes sure that calls to `fwrite` are executed atomically. To avoid additional locking, we first asynchronously collect complete proof lines in thread local buffers before calling `fwrite` and then rely on its implicit locking mechanism.

7 Experiments on Proofs

We want to evaluate the advantage of not copying the clauses in the proofs. But instead of implementing a variant that really copies clauses, we *fake copying*: the clauses are *only* duplicated in the proof, but not in the solver. We argue that this approach gives similar performance as really copying clauses, as it would be necessary in other multi-threaded solvers which copy clauses, both in terms of proof size and checking time.

However, adapting inprocessing was a challenge, as our current version of for instance bounded variable elimination during inprocessing requires that all irredundant clauses are deduplicated. Instead we *deactivated* any form of sequential inprocessing completely which requires deduplication. Thus, in the following experiments we only report on a variant of Gimsatul with initial preprocessing enabled, but only thread-local inprocessing enabled (vivification and failed literal probing).

We consider the 96 unsatisfiable problems that are solved by Gimsatul without inprocessing by all 1, 2, 4, 8, and 16-thread configurations (due to non-determinism, rerunning benchmarks could lead to a different set of solved problems though). We use unsatisfiable problems in order to be able to do backwards checking instead of forward checking as is required if the problem is satisfiable or no contradiction is derived. Due to the time required to check proofs ($\geq$10 h), we were not able to run the benchmarks for 32 threads before the submission deadline either.

Does Proof Generation Cause a Slowdown? Generating proofs is much more costly in a context of parallel SAT solving because proof logging is inherently single-threaded. However, according to our experiments in Figure 8, the cost of proof generation is negligible.

Do Proofs become Longer or Shorter? The answer to this question amounts to answering: is the work done by other rings useful or are the lines never used and could be removed. If the proofs are longer, then the work was useless. If the proof have similar length, then the rings just do work that has to be done too.

In Figure 9a we show the proof size for runs using different number of threads. The plots are not completely conclusive. The 2-thread configuration produces larger proofs than the 1-thread and 4-thread versions. We realized after looking at the results of the experiments that our portfolio strategy might not be the best possible one and explain the bad behavior of 2 and 16 threads compared to 4 or 1. For 2 threads, instead of using one thread that focuses on SAT (stable mode) and one that focuses on UNSAT (focused mode) following the idea of Chanseok Oh [13], one thread runs in focused mode and one in alternating mode. For 16 threads, we have 8 threads that are running the same strategy. Testing both assumptions is future work.

Is Sharing Clauses Useful? In Figure 9b, we show the length of the proofs without sharing the proofs. It is clear that the proofs are longer or even much longer. Interestingly, for most problems, the size does not change much between the 4-thread and the 2-thread version. This indicates that the number of exchanged clauses is either limited or that the shared clauses are

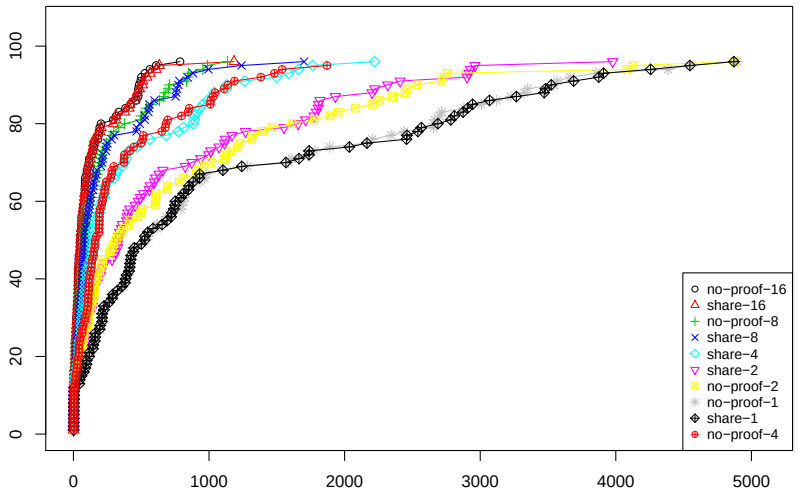
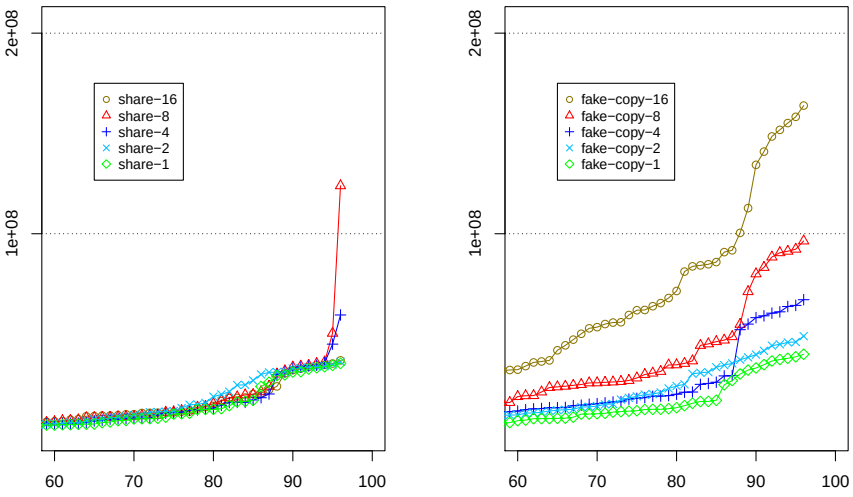


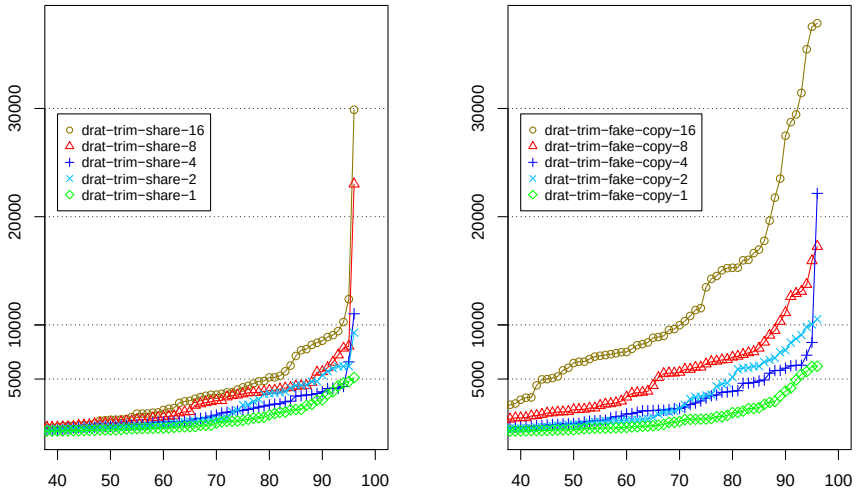
Figure 8: Cumulative distribution function (CDF) of the wall-clock solving time of Gimsatul with and without proof generation for the discussed variant with sequential inprocessing disabled



(a) Proof length with sharing

(b) Proof length with fake copying

Figure 9: Proof length



(a) Proof checking with sharing (b) Proof checking with fake copying

Figure 10: Amount of time required to check the proofs produced by GIMSATUL

useful to reduce the search space limiting the overhead: If every clause was shared and entirely useless, the proofs would be n times as large for n threads.

Is Proof Checking Easier? In Figure 10, we have plotted the amount of time DRAT-TRIM needed to check the proofs. It is important to notice that DRAT-TRIM does not seem to detect duplicates[3] and hence must reprove the lemma each time and every copy must be propagated when either of them is propagating.

With sharing, checking scales better, but is still much slower than in the single-threaded case. This can be explained by the fact that DRAT-TRIM does not know the context of learned clauses, i.e., which solver thread produced which clauses. The checker has to treat them all in the same way potentially increasing propagation across contexts, which arguably yields an overhead during checking redundancy.

Should Proof Checking be added to the SAT Competition? This is the most controversial question without clear answer: even without physical clause sharing, DRAT-TRIM is able to check clauses. However, even though checking 32-thread certificates is less than 32-times slower, it most likely is too slow to be run in practice. Thus we consider parallelizing DRAT checkers as an important future work to solve this problem. Alternatively we will look into producing proof formats with antecedent information, for which parallel checking is easier.

8 Conclusion

We presented the architecture of our newest solver GIMSATUL which shares clauses physically without copying. Even though it only features a simple form of diversification it scales linearly

[3]Our understanding of the source code of DRAT-TRIM is that it uses a hash table only to count the number of clauses for deletions, but not to avoid relearning clauses.

with the number of threads in our experiments. We also study the number of proof steps in this setting and observed that physical sharing yields smaller proofs. Nearly all proofs generated by runs with multiple solving threads have similar size as those produced by a single thread.

We want to further explore alternative clause exchange and search diversification strategies. We might also look into parallelizing variable elimination, subsumption and equivalent literal substitution, which are currently run by a single thread, even though this part does not seem to be a bottle-neck for large time-outs as used in the SAT competition. Making the solver deterministic like ManyGlucose would make the SAT solver easier to debug, and the biggest requirement, the time measurement by memory accesss, is already present in the code.

Acknowledgment. This work is supported by the Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), and the LIT AI Lab funded by the State of Upper Austria. Some of the design choices for the clause sharing architecture of Gimsatul are inspired by discussions with Christoph Kirsch and his former students Martin Aigner and Andreas Haas and other colleagues in RiSE [8]. The idea of using separate distributed pools inspired by k-stacks, in order to avoid a global lock-less queue and at the same time reduce false sharing was first proposed to us by Andreas in 2012 during our joint project. We also thank Sonja Gurtner for many timely textual improvements.

References

[1] Balyo, T., Sinz, C.: Parallel satisfiability. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 3–29. Springer (2018). doi:10.1007/978-3-319-63516-3_1

[2] Biere, A., Fleury, M.: Chasing target phases. In: Workshop on the Pragmatics of SAT 2020 (2020), http://fmv.jku.at/papers/BiereFleury-POS20.pdf

[3] Biere, A., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) SAT Competition 2021 (2021), submitted

[4] Biere, A., Järvisalo, M., Kiesl, B.: Preprocessing in SAT solving. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 391 – 435. IOS Press, 2nd edition edn. (2021)

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

[6] Frioux, L.L., Baarir, S., Sopena, J., Kordon, F.: Modular and efficient divide-and-conquer SAT solver on top of the Painless framework. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11427, pp. 135–151. Springer (2019). doi:10.1007/978-3-030-17462-0_8

[7] Hamadi, Y., Jabbour, S., Sais, L.: ManySat: solver description. Tech. Rep. MSR-TR-2008-83 (May 2008), https://www.microsoft.com/en-us/research/publication/manysat-solver-description/

[8] Henzinger, T.A., Kirsch, C.M., Payer, H., Sezgin, A., Sokolova, A.: Quantitative relaxation of concurrent data structures. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 317–328. ACM (2013). doi:10.1145/2429069.2429109, https://doi.org/10.1145/2429069.2429109

[9] Heule, M.J.: Marijn J.H. Heule, associate professor at carnegie mellon university (mar 2022), https://www.cs.cmu.edu/~mheule/, accessod on 31. May 2021.

[10] Järvisalo, M., Biere, A.: Reconstructing solutions after blocked clause elimination. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6175, pp. 340–345. Springer (2010). doi:10.1007/978-3-642-14186-7_30, https://doi.org/10.1007/978-3-642-14186-7_30

[11] Kaufmann, M., Kottler, S., Kaufmann, M., Kottler, S.: SArTagnan - a parallel portfolio SAT solver with lockless physical clause sharing. In: Workshop on the Pragmatics of SAT 2020 (2011)

[12] Mahajan, Y.S., Fu, Z., Malik, S.: zChaff 2004: An efficient SAT solver. In: Hoos, H.H., Mitchell, D.G. (eds.) Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3542, pp. 360–375. Springer (2004). doi:10.1007/11527695_27

[13] Oh, C.: 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. pp. 307–323 (2015). doi:10.1007/978-3-319-24318-4_23

[14] Prevot, N., Soos, M., Meel, K.S.: Leveraging GPUs for effective clause sharing in parallel SAT solving. In: Li, C., Manyà, F. (eds.) Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12831, pp. 471–487. Springer (2021). doi:10.1007/978-3-030-80223-3_32

[15] Rebola-Pardo, A., Biere, A.: Two flavors of DRAT. In: Berre, D.L., Järvisalo, M. (eds.) Proceedings of Pragmatics of SAT 2015, Austin, Texas, USA, September 23, 2015 / Pragmatics of SAT 2018, Oxford, UK, July 7, 2018. EPiC Series in Computing, vol. 59, pp. 94–110. EasyChair (2018). doi:10.29007/lt8r

[16] Schreiber, D., Sanders, P.: Scalable SAT solving in the cloud. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 518–534. Springer (2021). doi:10.1007/978-3-030-80223-3_35

[17] Schubert, T., Lewis, M., Becker, B.: PaMiraXT: Parallel SAT solving with threads and message passing. J. Satisf. Boolean Model. Comput. **6**(4), 203–222 (2009). doi:10.3233/sat190068, http://www.avacs.org/Publikationen/Open/schubert.jsat09.pdf

[18] Soos, M.: Memory layout of clauses in minisat (mar 2016), https://www.msoos.org/2016/03/memory-layout-of-clauses-in-minisat/, accessod on 31. May 2021

[19] Tan, Y.K., Heule, M.J.H., Myreen, M.O.: cake_lpr: Verified propagation redundancy checking in CakeML. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 223–241. Springer (2021). doi:10.1007/978-3-030-72013-1_12

[20] Vallade, V., Le Frioux, L., Oanea, R., Baarir, S., Sopena, J., Kordon, F., Nejati, S., Ganesh, V.: New concurrent and distributed painless solvers: P-MCOMSPS, P-MCOMSPS-COM, P-MCOMSPS-MPI, and P-MCOMSPS-COM-MPI. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) SAT Competition 2021 (2021), submitted

[21] Wetzler, N., Heule, M., Jr., W.A.H.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8561, pp. 422–429. Springer (2014). doi:10.1007/978-3-319-09284-3_31

Workshop Paper

[1] Florian Pollitt, Fleury, Mathias, and Armin Biere. "Efficient Proof Checking with LRAT in CaDiCaL (work in progress)". In: *24th GMM/ITG/GI Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems, MBMV 2023, Freiburg, Germany, March 23-23, 2023*. Ed. by Armin Biere and Daniel Große. I supervised Florian and wrote parts of the paper. VDE, 2023, pp. 64–67.

Efficient Proof Checking with LRAT in CaDiCaL (Work in Progress)

Florian Pollitt, Mathias Fleury, and Armin Biere
pollittf@informatik.uni-freiburg.de, fleury@cs.uni-freiburg.de, biere@cs.uni-freiburg.de
University of Freiburg, Germany

Abstract

The proof format DRAT used in the SAT competition is rather inefficient to check, often even slower to check than it takes the SAT solver to solve the instance and generate the proof. Therefore we implemented within the SAT solver CaDiCaL the possibility to generate LRAT proofs directly, where LRAT on the one hand is much easier and way more efficient to check, but on the other hand much harder to generate. Unlike previous approaches our implementation generates LRAT directly in the solver without intermediate translations. We further propose the tool LRAT-Trim, which can trim redundant proof steps from LRAT proofs, which not only reduces proof size but also leads to much faster proof checking.

1 Introduction

Proof checking is an important part of SAT solving, e.g., unsatisfiable problems do not count as solved in the SAT Competition unless a proof is provided which passes a proof checker. To increase trust even further proof checkers are used which are entirely verified [5, 7].

The currently only allowed proof format in the SAT Competition is DRAT [8].[1] The main issue with DRAT is that checking can take several times the amount of solving time. The reason is that the DRAT proof certificate format favors ease of generation and is not detailed enough to avoid searching during checking. Therefore both the solver and the checker have to propagate clauses (actually using similar data structures). To reduce this overhead (and simplify verification) all verified checkers follow the same principle. First the DRAT proof is converted by an (untrusted) external program to a more detailed proof format such as LRAT [5] or GRAT [7]. The resulting proof in an enriched format – containing enough details to avoid search – is checked by the verified program instead.

Note, however, that neither our SAT solver CaDiCaL [3] nor the winners of the SAT Competition of the last 2 years need the full power of RAT. They provide proofs in the upward-compatible but less powerful DRUP proof format. On the other hand CaDiCaL contains many different inprocessing techniques, which makes it a good candidate to implement direct generation of LRAT proofs – even though some of these techniques are not activated by default.

A similar attempt to resolve the performance issues with DRAT [1] lead to a new proof format, FRAT, that sits between LRAT (because it allows for justifications) and DRAT (because it still allows steps without justification). Their aim was to fill out most *gaps* and leave the *"harder"* cases as black box to be filled in by the proof checker. Therefore, they still use a tool Frat-Rs to convert the FRAT proof to a proper LRAT proof – trimming the proof on the way, to reduce the number of proof steps to check.

In this work-in-progress, we have extended our SAT solver CaDiCaL [3] to generate the richer LRAT format directly. The focus of our work is on three different aspects:

C1. directly produce correct LRAT proofs

C2. without slowing down the solver and

C3. without changing its search space.

Our first goal **C1** lead us to reimplement LRAT generation during conflict analysis and as part of all inprocessing techniques of CaDiCaL, some of which were not covered in the FRAT implementation, such as equivalent literal substitution (Section 3).

As it is common, our implementation generates a vast number of proof steps on-the-fly, from which however at the end a significant fraction turns out to be unnecessary to derive the empty clause $\bot$. This applies to most tools that process DRAT or FRAT which accordingly can benefit from some form of trimming. Therefore, we also implemented a tool called Lrat-Trim to trim proofs down and improve performance of checking proofs with the verified checker Cake_Lpr (Section 4).

At this point we can not report on extensive experiments yet, and therefore we focus in this work-in-progress report on making our proof generation robust and will discuss preliminary results for a problem with a very large large proof from the SAT Competition 2022. (Section 5).

Our implementation is publicly available[2] and is going to be merged into the main CaDiCaL version.

2 Preliminaries

For a detailed introduction to SAT solving, we refer to the *Handbook of Satisfiability* [4]. For the purpose of this paper, it is sufficient to know that SAT solvers build a partial

[1] A change was announced in the SAT Competition 2023, as different proof checkers (and therefore different proof format) could be allowed.

[2] `https://github.com/florianpollitt/radical`

```
p cnf 2 4
1 2 0
1 -2 0
-1 2 0
-1 -2 0
```
(a) DIMACS input

```
1 0
d 1 2 0
d 1 -2 0
2 0
d -1 2 0
0
```
(b) DRAT proof

```
o 1 1 2 0
o 2 1 -2 0
o 3 -1 2 0
o 4 -1 -2 0
a 5 1 0 1 1 2 0
d 1 1 2 0
d 2 1 -2 0
a 6 2 0
d 3 -1 2 0
a 7 0 1 5 6 4 0
f 4 -1 -2 0
f 5 1 0
f 6 2 0
f 7 0
```
(c) FRAT proof

```
5 1 0 1 2 0
5 d 1 2 0
6 2 0 5 3 0
6 d 3 0
7 0 5 6 4 0
```
(d) LRAT proof

Figure 1 Example DRAT, FRAT, and LRAT proofs for the same CNF on the left.

model. Along the way, they learn new clauses preserving satisfiability until either the partial model becomes a total model (translating the model back to a model of the original clause set) or the empty clause $\bot$ is derived, meaning that the problem is unsatisfiable.

The DRAT proof format consists simply of all the clauses learned by the SAT solver. This design decision helps DRAT to easily capture all techniques currently used by SAT solvers without the need to provide justification. The LRAT proof format provides more detailed information: Each clause gets an identifier and each step is the result of resolving several clauses together. The list of clauses is given as justification for each step. The last derived clause is $\bot$ – showing that the problem is unsatisfiable.

In related work [1] a new proof format was proposed that sits in-between LRAT and FRAT: some justification can be left out. This reduces the amount of implementation work in the SAT solver, since certain types of functions can be left unchanged, particularly if they infrequently contribute to the proof. Confusingly, the option to activate this proof is called `--lrat`. Throughout the rest of the paper, we will call this implementation (CaDiCaL) FRAT, even when we talk specifically about the generation of LRAT steps, because we have nothing to say about the other steps that are simply unchanged DRAT steps.

Figure 1 illustrates these proof formats for a simple example. The shortest proof is obviously the DRAT proof 1b, but it is missing information on how clauses were derived (in dark blue). In FRAT we have to repeat all the input assumptions, starting with o. The justification steps are also optional (see `a 6 2 0` without justification).

3 Implementation

Most of the actual computation can be done alongside the generation of clauses, including clause learning, which by definition consists of resolving clauses in the order given by the partial model (Section 3.1). However, some techniques require a deeper change like equivalence literal substitution (Section 3.2).

3.1 Conflict Analysis

Most clauses derived by a SAT solver originate from conflict analysis. When the solver finds a mismatch between the current partial assignment and the clauses, one conflicting clause is analyzed and the partial assignment adjusted.

One recent addition to the conflict analysis is based on the concept of "shrinking" [6]. The idea in shrinking is to derive the first unique implication point [4] on each level without increasing the proof size. This is very useful for problems with many binary clauses, such as the planning instances from the SAT Competition 2020.

Unlike FRAT proof generation for minimization, our implementation perfoms a post-processing step direclty on the learned clause instead of repropagation. To this extent we identify literals that were removed or added and add the necessary reason clauses as needed:

```
C_old := Clause before shrinking
C_new := Clause after shrinking
Chain_old := LRAT chain for C_old
Chain_new := empty LRAT chain

calculate_lrat_chain (literal K)
  C := reason of K in the current assignment
  foreach literal L in C different from K
     if not (reason of L in Chain_new and
             reason of L in Chain_old) and
        L not in C_new)
      calculate_lrat_chain (L)

  add C to Chain_new

for each literal L in C_old
  if L is not in C_new
    calculate_lrat_chain (L)

Chain_new := Chain_new + Chain_old
```

This works both for the standard minimization which is actually used inside shrinking as well as shrinking.

3.2 Equivalence Literal Substitution

Equivalent literal substitution is a procedure that detects and replaces equivalent literals by a chosen representative. For example, if the problem includes the clauses $\neg A \vee B$ and $A \vee \neg B$, we know that A and B are equivalent and we can replace all occurrences of either literal by the other.

We use Tarjan's algorithm to detect cycles in the graph spanned by the binary clauses and then fix a representative for each cycle. In the DRAT proof we simply dump all changed clauses and delete the old ones.

For LRAT we have to produce the resolution chain. This can only be calculated after fixing the representative and is done for each replacement in every clause separately, similarly to the process described in Section 3.1.

Fixing the representative is a rather arbitrary choice (smallest absolute value). We considered changing this to first visited by Tarjan's algorithm. This change would allow us to reuse some computation possibly making the generation of LRAT more efficient. In the end we decided against it to keep the behavior of CaDiCaL the same.

4 Trimming LRAT proofs

In early experiments we observed that the FRAT tool chain produced significantly smaller proofs, allowing for much more efficient proof checking. This is because clauses which are not required to derive the empty clause are trimmed from the proof and do not have to be checked nor at the end verified. An important feature of proof checking is the ability to trim down the proof which helps to reduce this redundant checking.

Even though trimming is very effective it is not obvious how to achieve this reduction in DRAT because dependencies between proof steps are missing. In LRAT these dependencies are listed explicitly and we implemented a proof trimmer called LRAT-TRIM to make use of this fact. It allows us to regularly achieve a reduction by a factor of 2 to 3 also again emphasizing how many useless clauses a SAT solver actually derives during search.

In essence, trimming is about doing a backward liveness analysis skipping clauses which are not useful. However, it is not possible to write a file backwards, so we only iterate over the graph starting from the $\perp$ clause at the end.

```
mark_antecedents(clause C)
   if C is marked return
   if C is an original clause return
   mark C as used
   for each antecedent D of C
     mark_antecedents (D)
```

Once the algorithm has identified all the useful proof steps, we can dump the proofs back to a file. One step we have not experimented with is the deletion of clauses: Studying whether it is better to immediately delete clauses or wait and delete several clauses at once is left as future work. However, we observed that eagerly deallocating removed clauses unfortunately does not improve performance, but it does reduce maximum memory usage substantially.

5 Early Experiments

After implementing LRAT production in our SAT solver CaDiCaL we first identified a minor necessary change (Section 5.1) that has no major impact. Besides fuzzing we have also tested our approach on input files containing unit clauses, which was not supported by FRAT (Section 5.2). Finally we report on the performance difference for a single problem with a very large proof (Section 5.3).

5.1 No Behavior Difference

During our experiments to validate goal **C3**, we realized that we had to change solver behavior in two ways. First, scheduling of garbage collection during bounded-variable elimination depends on the size of the clauses, which however changed with LRAT proof generation, as clauses became larger due to the additional required clause identifier `id` field. Our new version of CaDiCaL thus is always forced to use clause identifiers which however we do not consider to have a substantial impact on performance no memory usage. The second change became necessary due to the way how conflicts were derived in equivalence literal detection: instead of stopping on detection of such a conflict, we now simply continue and later propagate the literal in order to produce a proper LRAT proof.

5.2 Robustness by Fuzzing

Our goal **C1** of always being able to generate proofs was achieved by intensive fuzzing of our solver, proof generation and proof checking. We first attempted to do the same with the old implementation, but immediately experienced failing proofs, due to several reasons, including handling of unit clauses in the input proof file.

We also observed that resolution chains often listed the same clause several times. Reducing these occurrences can lead to a polynomial speedup, since justifying one literal can pull in several more clauses (e.g., if some of the literals have been removed by minimization).

5.3 Performance Loss

We have not yet studied the goal **C2** much. Early experiments indicate that our solver is slightly slower but that solving and proof checking is significantly faster. As example we consider the problem `sudoku-N30-10` from the SAT Competition 2022 [2]. To make the times comparable, we activated clause identifiers in the basic CaDiCaL version, ported the modification from the FRAT version to the newest CaDiCaL 1.5.2. The only real algorithmic difference is our shrinking of learned clauses [6], which we deactivated for the comparison instead of fixing the FRAT proof generation. All runs are without shrinking (as it is not supported by the FRAT version). Under these restrictions the comparison is fair, as runs produce exactly the same search behavior, including the same exact number of conflicts. The experiments were run on a desktop computer with an Intel i9-12900 with 128 GB RAM and hyperthreading on, except for the GRAT generation which was run on a machine with 2 TB because 128 GB were not enough.

CaDiCaL	Solving time	Proof size	Conversion tool	Conversion+ trimming	Trimmed proof size	Checking tool	Verified checking	Total
no proofs	4 770 s	-	-	-	-	-	-	4 770 s
DRAT	4 801 s	21 GB	DRAT-TRIM	5 639 s	13 GB	CAKE_LPR	812 s	11 252 s
DRAT	4 801 s	21 GB	GRAT (64 threads)	916 s	13 GB	GRATCHK	326 s	6 043 s
FRAT	5 349 s	78 GB	FRAT-RS	1 907 s	23 GB	CAKE_LPR	900 s	8 156 s
LRAT⋆	5 100 s	70 GB	-	-	-	CAKE_LPR	3 819 s	8 919 s
LRAT⋆	5 100 s	70 GB	LRAT-TRIM⋆	263 s	18 GB	CAKE_LPR	900 s	6 263 s

Table 1 Timing with different workflows on the `sudoku-N30-10` from the SAT Competition 2022. The ⋆ symbol indicates that the tool is a contribution of this work.

Table 1 provides the detailed timings. The DRAT proof conversion needs slightly more time than the proof production. LRAT proof generation has a cost of around 7%, but the proof checking is heavily reduced. Trimming the proof has a cost of 263 s, but reduces the checking time by a factor 3 and the proof size by a factor 4.

One reason explaining that directly produced LRAT proofs are larger than translated proofs comes from a heuristic of the translation tools: If they have the choice between two clauses, they will pick the clause that has already been used, while our proof production will pick the one used internally to derive a clause.

6 Conclusion

We have implemented LRAT proof production in our SAT solver CaDiCaL. Early experiments show that performance is slightly reduced, but the full workflow of producing *and* checking the proof becomes much faster thanks to our other tool Lrat-Trim.

Future work includes a proper evaluation of the implementation on the entire set of problems of the SAT competition. Another interesting idea is to check the proofs online, directly while generated.

7 Literatur

[1] S. Baek, M. Carneiro, and M. J. H. Heule. A flexible proof format for SAT solver-elaborator communication. *Log. Methods Comput. Sci.*, 18(2), 2022. doi: 10.46298/lmcs-18(2:3)2022.

[2] T. Balyo, M. Heule, M. Järvisalo, M. Iser, and M. Suda, editors. *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*. Department of Computer Science, University of Helsinki, Finland, 2022. URL `https://helda.helsinki.fi/bitstream/handle/10138/347211/sc2022_proceedings.pdf`.

[3] A. Biere, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021. In M. Heule, M. Järvisalo, and M. Suda, editors, *SAT Competition 2021*, 2021.

[4] A. Biere, M. Järvisalo, and B. Kiesl. Preprocessing in SAT solving. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391 – 435. IOS Press, 2nd edition edition, 2021.

[5] L. Cruz-Filipe, M. J. H. Heule, J. Hunt, Warren A., M. Kaufmann, and P. Schneider-Kamp. Efficient certified RAT verification. In L. de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017. doi: 10.1007/978-3-319-63046-5_14.

[6] M. Fleury and A. Biere. Efficient All-UIP learned clause minimization. In C. Li and F. Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2021. doi: 10.1007/978-3-030-80223-3_12.

[7] P. Lammich. Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.*, 64(3):513–532, 2020. doi: 10.1007/s10817-019-09525-z.

[8] N. Wetzler, M. Heule, and J. Hunt, Warren A. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. doi: 10.1007/978-3-319-09284-3_31.

Conference Paper

[1] Florian Pollitt, Fleury, Mathias, and Armin Biere. "Faster LRAT Checking than Solving with CaDiCaL". In: *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*. Ed. by Luca Pulina, Laura Pandolfo, and Dario Guidotti. Vol. 271. LIPIcs. I supervised Florian and wrote parts of the paper. 2023, 21:1–21:12.

Faster LRAT Checking Than Solving with CaDiCaL

Florian Pollitt
Universität Freiburg, Germany

Mathias Fleury
Universität Freiburg, Germany

Armin Biere
Universität Freiburg, Germany

Abstract

DRAT is the standard proof format used in the SAT Competition. It is easy to generate but checking proofs often takes even more time than solving the problem. An alternative is to use the LRAT proof system. While LRAT is easier and way more efficient to check, it is more complex to generate directly. Due to this complexity LRAT is not supported natively by any state-of-the-art SAT solver. Therefore Carneiro and Heule proposed the mixed proof format FRAT which still suffers from costly intermediate translation. We present an extension to the state-of-the-art solver CaDiCaL which is able to generate LRAT natively for all procedures implemented in CaDiCaL. We further present Lrat-Trim, a tool which not only trims and checks LRAT proofs in both ASCII and binary format but also produces clausal cores and has been tested thoroughly. Our experiments on recent competition benchmarks show that our approach reduces time of proof generation and certification substantially compared to competing approaches using intermediate DRAT or FRAT proofs.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases SAT solving, Proof Checking, DRAT, LRAT, FRAT

Digital Object Identifier 10.4230/LIPIcs.SAT.2023.21

Supplementary Material
Dataset (Log files): `https://cca.informatik.uni-freiburg.de/lrat/` [20]

Acknowledgements We thank reviewers of SAT'23 and MBMV'23 for their detailed comments as well as Mario Carneiro for making FRAT-rs publicly available.

1 Introduction

Proof production became an essential part in SAT solving. For instance, unsatisfiable problems only count as solved in the SAT Competition if a certifiable proof is provided. Proofs do increase trust in solving results by providing certificates that can be checked independently. To increase trust even further proof checkers can also be entirely verified [6, 16].

In the past the only format allowed in the SAT Competition was DRAT [23], even though the SAT Competition 2023 announced to allow additional formats. However, checking DRAT proofs often takes several times the amount of solving time. The problem with DRAT is that the format is not detailed enough to avoid search during checking. Both the solver and the checker have to propagate clauses (actually using similar data structures). To reduce this overhead (and simplify verification) all verified proof checkers expect an enriched format. The DRAT proof is augmented and converted by an (untrusted) external program into such an enriched format, e.g., LRAT [6] or GRAT [16], which contains enough information to avoid search and can then be checked easily by the verified proof checker.

On top of the actual clause contents (its literals) the LRAT [6] format requires the following additional information: (*i*) clause identifiers (ids) are used to reference clauses and to make clause deletion steps more concise; (*ii*) clause antecedent ids used in the resolution chain when deriving an added clause through reverse unit propagation (RUP) [12], i.e., as

26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023).
Editors: Meena Mahajan and Friedrich Slivovsky; Article No. 21; pp. 21:1–21:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

asymmetric tautology (AT) [14]; (*iii*) the ID and further resolution paths to refute the resolvent of the added clause with all clauses containing a RAT (blocking) literal in case the added clause relies on the stronger resolution asymmetric tautology (RAT) property [15].

These RAT literals would be needed to model more powerful reasoning (such as blocked clause addition or symmetry breaking etc.) but neither our SAT solver CaDiCaL [4] nor any top performing SAT solver in the SAT Competition over the last 2 years actually used such reasoning. Therefore, our efforts to extend CaDiCaL did not need to address the full power of RAT and we can focus on producing "LRUP" proofs, i.e., reverse-unit-propagation (RUP) proofs, but still need to augment these proofs with ids and resolution chains.

A similar attempt [1] by Carneiro and Heule led to a new proof format, FRAT, that sits between LRAT (because it allows for justifications) and DRAT (because it still allows steps without justification). Their aim was to fill out most "*gaps*" and leave *"harder"* to implement cases as black box to be filled in by an (untrusted) proof checker, i.e., by their FRAT-rs tool used to convert an FRAT proof to a fully justified LRAT proof. In a recent paper [18] this limitation of the FRAT producing CaDiCaL [1] forced a parallel proof-producing version of the award winning SAT solver Mallob to deactivate all steps not covered by FRAT, i.e., most inprocessing, as native LRAT proof generation is needed.

In this tool paper, we present an extension of our SAT solver CaDiCaL [4] to generate the richer LRAT format directly. Our focus is on three different aspects: (A) producing LRAT proofs for all solver configurations on all benchmarks, (B) comparable performance and, further, (C) making sure the solver behaves the same with/without proof generation.

Our goal (A) lead us to reimplement LRAT generation in the conflict analysis and all inprocessing techniques of CaDiCaL, some of which were not covered in the FRAT [1] producing implementation, such as equivalent literal subsumption (Section 3).

Like other SAT solvers, CaDiCaL generates a vast number of proof steps from which at the end, a significant fraction turns out to be unnecessary for the derivation of the empty clause. Thus most tools that process DRAT or FRAT will trim these unnecessary steps from the proof. However, we are not aware of a tool that does this for LRAT. Therefore we implemented a new tool called Lrat-Trim to trim proofs down and improve the performance of checking the proof with the verified checker Cake_Lpr [21] (Section 4).

To validate robustness of our approach we extended CaDiCaL to internally check LRAT proofs too and fuzzed the extended solver. This allowed us to use the model-based tester Mobical (which comes with CaDiCaL) to find, debug, and fix bugs much more efficiently. We further ran the extended new solver on the unsatisfiable problems from the SAT Competition 2022. We observed (almost) no slow-down without proof production (0.3%) and only a small slow-down for producing LRAT (5%). Proof checking performance was improved considerably compared to the two competing approaches DRAT and FRAT (see Section 5). Checking (and producing) our LRAT proofs has an overhead of 30% over pure solving, compared to 125% for FRAT and 180% in the SAT Competition mode (i.e., slower than producing them). Without negligible overhead over plain solving with CaDiCaL, we managed to check proofs faster than they are produced for a state-of-the-art SAT solver.

Our CaDiCaL extension is available at `https://github.com/florianpollitt/radical` and will shortly be merged into the main CaDiCaL repository. Note that a preliminary version of this paper was presented at the MBMV workshop [19] as work in progress. Compared to that shorter version, we have improved and present Lrat-Trim, give an extensive evaluation on the entire problem set of the SAT Competition 2022 (not just a single problem) and in general provide more details on the implementation.

2 Preliminaries

For an introduction to SAT solving please refer to the *Handbook of Satisfiability* [5]. In our context it is sufficient to recall that SAT solvers build a partial assignment and along the way learn new clauses preserving satisfiability until either the assignment satisfies all clauses or the empty clause is derived, meaning that the problem is unsatisfiable.

A DRAT [23] proof is the sequence of all clauses learned (or in general deduced) by the SAT solver interleaved with clause deletion steps, which are used to help the proof checker to focus on the same clauses the solver would see at this point of the proof. This design principle helps DRAT [23] to easily capture all techniques currently used by SAT solvers without the need to provide more complex justification e.g. in the form of resolution chains.

The LRAT [6] proof format has more detailed information: Each clause is associated with a clause identifier and claimed to be the result of resolving/propagating several clauses in the given order. The list of antecedent clause ids forms a justification and is part of such an addition step in LRAT. In the rest of the paper we focus on finding these justification.

3 Implementation

The LRAT extension to CaDiCaL was implemented by the first author as part of his master project and proceeded in four stages: First, the internal proof checker in CaDiCaL for DRAT clauses was extended to produce LRAT proofs, which is quite inefficient but can still be enabled through the `--lrat-external` option. Second, a separate internal LRAT checker was added to CaDiCaL to validate proofs on-the-fly while running the solver. Third, we implemented LRAT production for CaDiCaL without any inprocessing. Finally, all different inprocessing techniques were instrumented to generate LRAT proof chains directly. Thanks to the second stage, proofs could be validated on-the-fly, dramatically reducing the implementation effort (particularly for debugging). The implementation of these four stages took around two months in total but the last two stages only two weeks.

The resolution chain for justifying a new clause can be computed alongside normal CDCL search with little computational overhead but clause minimization and shrinking are a bit more involved (Section 3.1). Proof production in preprocessing and inprocessing were of varying degree of difficulty. The most interesting inprocessing technique from this point of view is equivalent literal substitution which we discuss in Section 3.2.

3.1 Conflict Analysis

Most clauses derived by a SAT solver originate from clauses learned during conflict analysis. When the solver finds a mismatch between the current partial assignment and the clauses, i.e., a conflicting clause which is falsified, then this conflict is analyzed and a clause is learned which forces the solver to adjust the partial assignment. In the standard implementation of conflict analysis the learned clause is derived by resolving individual reason clauses in reverse assignment order, starting with the conflicting clause, which in turn immediately gives the necessary justification for the (non-minimized first UIP [24]) learned clause.

We have adapted our code to generate chains for various technique relying on conflict analysis such as hyper binary resolution [13] and vivification [17]. It is crucial to distinguish between techniques that eliminate false literals (thus, necessitating an extension of the proof chain) and those that do not.

One recent addition to improve conflict analysis is the concept of "shrinking" [9, 10] which can be interpreted as a more advanced version of "minimization" [8]. Minimization only removes literals from the learned clause following resolution paths in the implication graph,

but does not add any literals. The additional idea in shrinking is to continue trying to resolve literals on a particular decision level until all but one (the first UIP on that level) is left, however, without being allowed to add literals from a lower decision level.

Our approach differs from the FRAT flow [1]. Their solver performs a post-process analysis of the final learned clause $C_{mini+shrink}$ to rediscover the necessary propagation by traversing the implication graph, which repeats conflict analysis work. In contrast, we split the justification process into two parts. First, we derive the justification for the clause C_{UIP} alongside conflict analysis with little to no overhead. Then, we derive the missing resolution steps between C_{UIP} and the shrunken and minimized clause $C_{mini+shrink}$ as a post-process analysis. We identify literals that differ and add the required reason clauses. Although we still traverse parts of the implication graph, we avoid repeating the conflict analysis.

Our Algorithm 1 shows the postprocessing step only. The first step has already derived the justification $Chain_{UIP}$ for the first UIP clause $C_{original}$ from conflict analysis. Our postprocessing step calculates the justification chain in $Chain_{mini+shrink}$. For each removed literal L (in $C_{original}$ but not in $C_{shrunken}$), we extend the chain with additional justification steps (Line 3).

The function calculate_LRAT_Chain(L) (Line 5) extends the chains with the required reason and preserves the resolution order. It goes recursively over all literals of the reasons and extends the chain with the reason. If the function reaches a previously used reason (*already_added*), it can stop the analysis to avoid duplicated reasons in the chain. Our calculation stops when we reach literals that appear in $C_{shrunken}$ ($L \notin Chain_{new}$). After calculating the justification chain for minimization and shrink, we merge the two chains $Chain_{UIP}$ and $Chain_{new}$ (Line 4). Starting with an empty chain provides a valid proof when removing unit literals during both phases.

Algorithm 1 Recursively calculating the prefix LRAT chain for shrinking and minimizing.

Data: currently build LRAT chain $Chain_{UIP}$
Data: the clause before $C_{original}$ and after minimization and shrinking $C_{shrunken}$
Result: resulting LRAT chain $Chain_{full}$

```
1  foreach literal L in C_original do
2      if L not in C_shrunken then
3          calculate_LRAT_Chain(L)
4  Chain_full := Chain_mini+shrink + Chain_UIP
5  calculate_LRAT_Chain (Literal K)
6      C := reason of K in the current assignment
7      foreach Literal L in C different from K do
8          already_added := reason of L in Chain_mini+shrink
9          if ¬already_added and L ∉ C_shrunken then
10             calculate_LRAT_Chain(L)
11     append C to Chain_mini+shrink
```

Our approach can potentially lead to duplicated unit clauses: We add unit clauses to the chain during conflict analysis. We can guarantee no duplicates here, but the same unit clause might also be added during post process analysis, which means it is actually needed earlier in $Chain_{mini+shrink}$ and we could remove it from $Chain_{UIP}$. Note that this cannot happen for larger clauses since they can appear at most once as a reason for some assignment. Since removing these unit clauses afterwards would be rather costly, we actually collect unit clauses separately and put them at the start of the merged chain after the post process analysis for $C_{shrunken}$ is finished. Like this, we can avoid duplicates and still get a correct justification chain for $C_{shrunken}$.

3.2 Equivalence Literal Substitution

While the justification process for clauses derived during variable elimination and other preprocessing techniques that rely on propagation and conflict analysis is similar to normal learning, producing LRAT proof justifications for equivalent literal substitution [5] is more involved.

Equivalent literal substitution detects and replaces equivalent literals by a chosen representative. For example, if the problem includes the three clauses $(\neg A \vee B)$, $(\neg B \vee C)$ and $(\neg C \vee A)$ we know that A, B and C are equivalent and we can replace all occurrences of either literal by one of the others. As is common we use Tarjan's algorithm [22] to detect cycles in the graph spanned by the binary clauses (i.e., the binary implication graph) and fix a representative for each cycle [5]. In the DRAT proof we can simply dump all changed clauses and delete the old ones.

For LRAT we have to produce the resolution chains. After fixing representatives, proof chains have to be produced for every changed clause separately. We derive the justification for each changed or removed literal, similarly as for the shrunken clause in conflict analysis 3.1.

Fixing the representative is a rather arbitrary choice (the smallest absolute value in this implementation). We considered changing this to the first visited literal during DFS in Tarjan's Algorithm, in order to allow reusing some computation and potentially shorten proofs, but in the end decided against changing solver behavior.

4 Trimming LRAT proofs

In preliminary experiments we observed that the FRAT flow [1] produced significantly smaller proofs. FRAT-rs trims the proof during translation to LRAT, i.e., it omits clauses that are not needed to derive the empty clause, allowing for much more efficient proof checking. We concluded that we needed a tool to do such trimming on LRAT directly in order to obtain an efficient pure LRAT proof generation and checking flow.

Even though trimming is effective, it is not obvious how to cheaply achieve such reduction for DRAT proofs because dependencies between proof steps are lacking. Luckily, in LRAT these dependencies are explicit. Therefore we implemented Lrat-Trim [2], an open-source LRAT proof trimming and checking tool. It often reduces proofs by a factor of 2 to 3, again emphasizing how many useless clauses a SAT solver actually derives during search.

Trimming LRAT proofs consists of a backward reachability analysis starting from the empty clause towards the clauses of the original CNF, marking reached clauses as needed. Clauses unmarked after this traversal are redundant and can be trimmed. This algorithm is implemented by depth first search (DFS) along antecedent clauses in justification chains.

It also determines the last usage of each clause ID and remaps original clause ids to a consecutive ID range. On completion we can dump the proofs back to a file in a forward manner, only writing needed clauses and their antecedents and skipping redundant clauses. While doing this we can eagerly mark clauses once they are not used anymore.

Before starting to write proof lines, we check whether there are redundant original clauses and if so write a single deletion line with all unused original clause ids. This minimizes the life-span of clauses in the trimmed LRAT proof, both for added and original clauses. Note that Lrat-Trim, in contrast to DRAT-Trim, does not require access to the original CNF nor looks at literals of clauses to trim proofs.

We also implemented a checking mode in Lrat-Trim which, given the original CNF and an LRAT proof, checks that the resolution chains of added clauses can be resolved to produce the claimed clauses. It also checks that clauses are not used after they are deleted in a deletion step. This checking mode comes in two flavors. The default is to first trim the clauses with the trimming algorithm described above and only check needed clauses. Alternatively Lrat-Trim supports forward checking, which checks added clauses on-the-fly during parsing and in particular allows to delete clauses in deletion steps eagerly.

On the one hand, forward checking reduces maximum memory usage to at most that of the solving process, whereas backward checking needs to keep the whole proof in memory which is usually much more than maximum usage during solving. On the other hand, forward checking substantially increases checking time, as all clauses have to be checked without trimming information, irrespective of being needed or not.

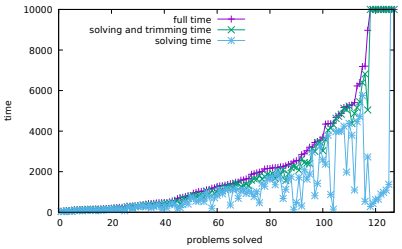
During the development of Lrat-Trim substantial effort went into making parsing as fast and robust as possible and also provide meaningful error messages during parsing and checking. The parsing code amounts to roughly 900 lines of C code out of 2400 lines for the whole tool (including comments but formatted with ClangFormat).

All three proof formats (DRAT, FRAT and LRAT) have a binary version. We implemented the binary format for LRAT (both in CaDiCaL and in Lrat-Trim) which is only supported by CLRAT [7], a formally verified checker for LRAT using ACL2. We are grateful to Peter Lammich who provided us a tool that converts LRAT proofs (with some extra requirements on proofs) to GRAT [16] that his checker can check. However, GRAT is stricter as duplicate or extraneous ids are not allowed. We leave it to future work to produce stricter proofs.

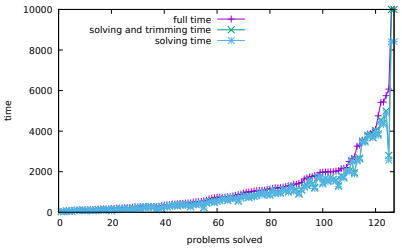
5 Experiments

While checking for our extensions not to change solver behavior with and without proof generation, i.e., validating (C), we realized that two changes to the solver became necessary. First, scheduling of garbage collection during bounded-variable elimination depends on the number of bytes allocated for clauses, which changed with LRAT proof generation, as clauses require an ID and thus became larger. Therefore, our CaDiCaL extension always uses clause ids, which is not expected to have major impact on performance nor memory usage. The second change is due to the way conflicts were derived in equivalent literal detection. Originally detection was aborted on such a conflict, which we now simply delay until detection finishes. Then the conflicting literal is propagated to yield a proper LRAT proof.

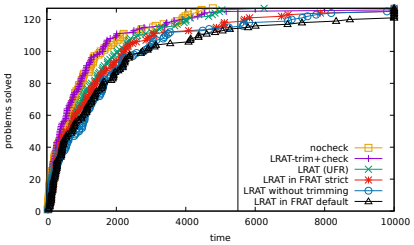
Our goal (A) of being able to always generate correct proofs was tested by intensive fuzzing of our solver, proof generation, and proof checking. We attempted to apply the same approach to the FRAT extension of CaDiCaL [1] but immediately experienced failing proofs, due to several reasons, particularly with respect to handling unit clauses in the input



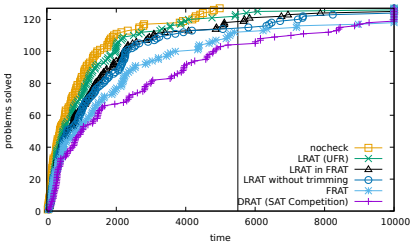
(a) Checking FRAT proofs of our new CaDiCaL version 1.5.1 from UFR but in the configuration of CaDiCaL version 1.2.1 used in [1].



(b) Checking LRAT proofs of our new CaDiCaL version 1.5.1 from UFR but in the configuration of CaDiCaL version 1.2.1 used in [1].



(c) Comparing trimmers Lrat-Trim vs. FRAT-rs (LRAT in FRAT) using CaDiCaL 1.5.1 proofs before checking with Cake_Lpr, except for the run "LRAT-trim+check" which checks with Lrat-Trim.



(d) CDF of all the solving and checking flows with the vertical black line indicating the 5,000 seconds timeout used for the solver, showing that our flow is the fastest with fewer timeouts.

Figure 1 Performance on unsatisfiable instances from the SAT Competition 2022.

CNF. We also observed that chains often listed the same clause id multiple times. Reducing these occurrences might lead to a substantial speedup, since justifying one literal can pull in several more clauses (e.g., if some of the literals have been removed by minimization).

After fuzzing, we ran our LRAT flow on the problems of the SAT Competition 2022 and found three issues: (*i*) Cake_Lpr did not accept some input files, because they contained trailing empty lines, which we then removed manually; (*ii*) Cake_Lpr requires a very large amount of memory (around the size of the proof file); (*iii*) one node of the cluster showed irregular behavior, when many proofs were written to the temporary disk at the same time, which lead to corrupted proof files resulting in an Lrat-Trim error. Reducing the number of jobs per node fixed this issue and we did not discover any further problem with the generated proofs, validating (A) and showing again the effectiveness of fuzzing.

To compare performance, i.e., showing that we achieved (B), of our extended version to the base version of CaDiCaL (added clause ids taking up space without being used), we let both versions write generated proofs to `/dev/null` in order to ensure that we do not introduce any bias due to file I/O limits as LRAT proofs exceed DRAT proofs in size substantially. This yielded an average overhead of 5% for our new LRAT proof production versus DRAT in base CaDiCaL.

For the remaining empirical analysis we have chosen to focus on the 127 benchmarks from the SAT Competition 2022, which were shown to be unsatisfiable during the competition. First, we tried to determine how much proofs can be reduced with our new tool Lrat-Trim.

It turns out, that some proofs were reduced to *one percent*, i.e., 99% of the output is not useful for deriving the contradiction. These problems stem from the `sudoku-N30` family. In other proofs 80% and more clauses are needed – most of these problems have a short runtime (around 200 s), contain a large amount of fixed variables and accordingly many clauses are simplified by removing these units, where each removal contributes a proof step.

In order to determine the performance of our new solving and checking flow, we compared the following three workflows: (*i*) the (competition) DRAT workflow, i.e., generating the DRAT proof, converting it to LRAT with DRAT-Trim, then checking that proof; (*ii*) the FRAT workflow, i.e., generating the FRAT proof, converting it to LRAT with FRAT-rs, then checking it; (*iii*) our new LRAT flow including generating, trimming, and checking the proof. All workflows use binary proof formats, except for feeding Cake_Lpr at the end.

We also ported the FRAT extensions [1] to the newest CaDiCaL version, but did not try to fix any issues. Nevertheless, we ran the ported version (see Figure 1a) which is now able to use the latest heuristics used in CaDiCaL, except for shrinking which had to be deactivated as it is not supported by the original FRAT code [1].

The first observation we can make is that the overhead of trimming and proof checking is quite consistent among our configurations, but wildly differs for FRAT: If many clauses without justification are used for the proof, the translation needs a lot of search – although, as expected, less than using the conversion to DRAT

To our surprise, we observed several timeouts though. They all seem to origin from one family submitted by AWS in 2022, where solving took less than 600 s, but elaboration (translation) never finishes. In comparison, DRAT-Trim also needs a very long time (6 000 s), but stays well below the time limit. It is unclear what the problem is and thus we tested one instance `aws-c-common:aws_priority_queue_s_sift_either` on a (twice as fast) computer where it took nearly 10 h to convert the 400 MB FRAT proof to a 3.8 GB LRAT proof. We have reported the issue on GitHub,[1] but have not heard back yet.

A comparison of Lrat-Trim with FRAT-rs in both *normal mode* and *super strict mode* is shown in Figure 1c. We used the feature of our extended version of CaDiCaL to generate proofs both in LRAT and in FRAT, where in FRAT, every step is properly justified. The results show that Lrat-Trim scales much better than FRAT-rs, although there was a bug which we reported that made FRAT-rs significantly slower when not using the *super strict mode*. Furthermore, Lrat-Trim can also check proofs directly and it turns out that the additional overhead of this (untrusted) checking compared to parsing and trimming is small.

Overall, our new LRAT proof flow performs best, with reasonably small overhead on solving. To ease visual comparison, we printed all different configurations into a single graph (Figure 1d). The fastest option is (of course) "no-checking" but our new method is not too far behind. Figure 2 shows that the overhead (cost) of proof checking compared to not checking any proofs. Our approach performs best taking only 30% more time than pure solving. The existing competing approaches are much slower with DRAT incurring an overhead of 180% and FRAT still requiring 125% more time than solving, i.e., both more than doubling overall certification time, while our approach has faster checking than solving.

As a sanity check, we also tested our LRAT proof flow using the default shrinking (see Fig 3). We observed that our new approach remains faster compared to the FRAT proof flow, confirming our initial findings.

[1] `https://github.com/digama0/frat/issues/18`

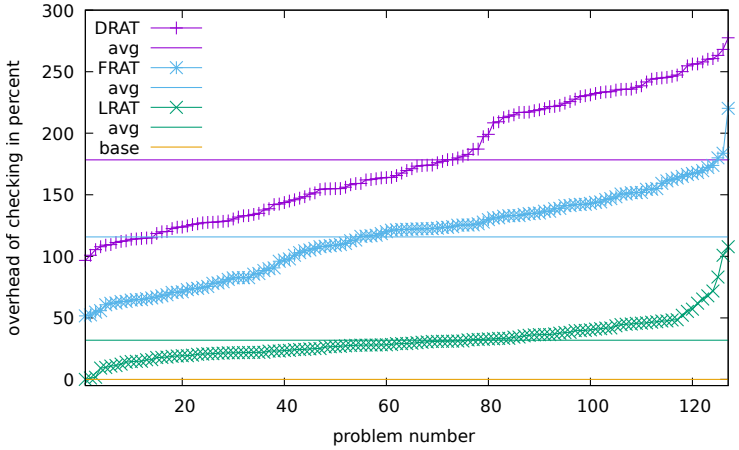


Figure 2 Overhead of the whole checking flow using FRAT, DRAT and our new LRAT flow on top of plain solving (without proof generation and checking), with averages shown as horizontal lines (LRAT 30% overhead, FRAT 125% and DRAT 180% overhead).

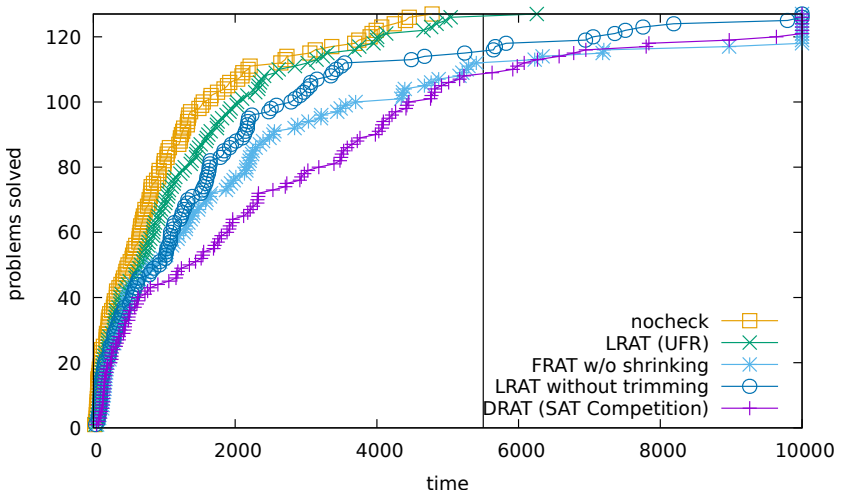


Figure 3 CDF all methods with vertical line indicating timeout of the solver with default options (i.e., with shrinking not supported by the CaDiCaL).

6 Conclusion

We have implemented native LRAT proof production in our SAT solver CaDiCaL. Even though direct production of LRAT proofs slows down the solver slightly this loss is by far offset by the reduction in proof checking time, both compared to DRAT and FRAT proofs. At the end our certification flow adds only 30% overhead compared to pure solving while other approaches take more than twice the time for certification.

It might be interesting to apply this work to recent results on distributed proof generation in the context of the cloud solver Mallob [18] as well as our multi-core solver in Gimsatul [11]. We also see the question of how to handle clause ids for virtual binary clauses as a technical challenge. Such clauses occur in both Gimsatul [11] and the state-of-the-art sequential solver kissat [3].

References

1 Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. *Log. Methods Comput. Sci.*, 18(2), 2022. doi:10.46298/lmcs-18(2:3)2022.

2 Armin Biere. Lrat trimmer, Last access, March 2023. Source code. URL: https://github.com/arminbiere/lrat-trim.

3 Armin Biere and Mathias Fleury. Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*, pages 10–11. University of Helsinki, 2022.

4 Armin Biere, Mathias Fleury, and Mathias Heisinger. CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021. In Marijn J. H. Heule, Matti Järvisalo, and Martin Suda, editors, *SAT Competition 2021*, 2021.

5 Armin Biere, Matti Järvisalo, and Bejamin Kiesl. Preprocessing in SAT solving. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391–435. IOS Press, 2nd edition edition, 2021.

6 Luís Cruz-Filipe, Marijn J. H. Heule, Jr. Hunt, Warren A., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction – CADE 26 – 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017. doi:10.1007/978-3-319-63046-5_14.

7 Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 220–236, Cham, 2017. Springer International Publishing.

8 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

9 Nick Feng and Fahiem Bacchus. Clause size reduction with all-UIP learning. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020 – 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 28–45. Springer, 2020. doi:10.1007/978-3-030-51825-7_3.

10 Mathias Fleury and Armin Biere. Efficient All-UIP learned clause minimization. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021 – 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2021. doi:10.1007/978-3-030-80223-3_12.

11 Mathias Fleury and Armin Biere. Scalable proof producing multi-threaded SAT solving with Gimsatul through sharing instead of copying clauses. *CoRR*, abs/2207.13577, 2022. doi:10.48550/arXiv.2207.13577.

12 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008. URL: http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf.

13 Marijn Heule, Matti Järvisalo, and Armin Biere. Revisiting hyper binary resolution. In Carla P. Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2013. doi:10.1007/978-3-642-38171-3_6.

14 Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for CNF formulas. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning – 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2010. doi:10.1007/978-3-642-16242-8_26.

15 Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning – 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012. doi:10.1007/978-3-642-31365-3_28.

16 Peter Lammich. Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.*, 64(3):513–532, 2020. doi:10.1007/s10817-019-09525-z.

17 Chu-Min Li, Fan Xiao, Mao Luo, Felip Manyà, Zhipeng Lü, and Yu Li. Clause vivification by unit propagation in CDCL SAT solvers. *Artif. Intell.*, 279, 2020. doi:10.1016/j.artint.2019.103197.

18 Dawn Michaelson, Dominik Schreiber, Marijn J. Heule Heule, Benjamin Kiesl-Reiter, and Michael W. Whalen. Unsatisfiability proofs for distributed clause-sharing SAT solvers. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 28th International Conference, TACAS 2023*, Lecture Notes in Computer Science, 2023. Accepted, to appear.

19 Florian Pollitt, Mathias Fleury, and Armin Biere. Efficient proof checking with lrat in cadical (work in progress). In Armin Biere and Daniel Große, editors, *24th GMM/ITG/GI Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems, MBMV 2023, Freiburg, Germany, March 23-23, 2023*, pages 64–67. VDE, 2023. Accepted. URL: https://cca.informatik.uni-freiburg.de/papers/PolittFleuryBiere-MBMV23.pdf.

20 Florian Pollitt, Mathias Fleury, and Armin Biere. Native LRAT in CaDiCaL for faster proof checking, 2023. URL: https://cca.informatik.uni-freiburg.de/lrat.

21 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake_lpr: Verified propagation redundancy checking in cakeml. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 223–241. Springer, 2021. doi:10.1007/978-3-030-72013-1_12.

22 Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.

23 Nathan Wetzler, Marijn J. H. Heule, and Jr. Hunt, Warren A. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014 – 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. doi:10.1007/978-3-319-09284-3_31.

24 Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in Boolean satisfiability solver. In Rolf Ernst, editor, *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001*, pages 279–285. IEEE Computer Society, 2001. doi:10.1109/ICCAD.2001.968634.

Journal Paper

[1] Daniela Kaufmann, Fleury, Mathias, Armin Biere, and Manuel Kauers. “Practical Algebraic Calculus and Nullstellensatz with the Checkers Pacheck and Pastèque and Nuss-Checker”. In: *Formal Methods in System Design*. I did the entire work Isabelle and the details of the side-conditions where done together to find out why I was having issues in the proof. 2022.

Formal Methods in System Design manuscript No.
(will be inserted by the editor)

Practical Algebraic Calculus and Nullstellensatz with the Checkers Pacheck and Pastèque and Nuss-Checker

Daniela Kaufmann · Mathias Fleury · Armin Biere · Manuel Kauers

the date of receipt and acceptance should be inserted later

Abstract Automated reasoning techniques based on computer algebra have seen renewed interest in recent years and are for example heavily used in formal verification of arithmetic circuits. However, the verification process might contain errors. Generating and checking proof certificates is important to increase the trust in automated reasoning tools. For algebraic reasoning, two proof systems, Nullstellensatz and polynomial calculus, are available and are well-known in proof complexity. A Nullstellensatz proof captures whether a polynomial can be represented as a linear combination of a given set of polynomials by providing the co-factors of the linear combination. Proofs in polynomial calculus dynamically capture that a polynomial can be derived from a given set of polynomials using algebraic ideal theory. In this article we present the practical algebraic calculus as an instantiation of the polynomial calculus that can be checked efficiently. We further modify the practical algebraic calculus and gain LPAC (practical algebraic calculus + linear combinations) that includes linear combinations. In this way we are not only able to represent both Nullstellensatz and polynomial calculus proofs, but we are also able to blend both proof formats. Furthermore, we introduce extension rules to simulate essential rewriting techniques required in practice. For efficiency we also make use of indices for existing polynomials and include deletion

This work is supported by the LIT AI Lab funded by the State of Upper Austria and by Austrian Science Fund (FWF) P31571-N32.

Daniela Kaufmann
Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria
E-mail: daniela.kaufmann@jku.at

Mathias Fleury
Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria
E-mail: mathias.fleury@jku.at

Armin Biere
Chair of Computer Architecture, Albert-Ludwigs-University, Freiburg, Germany
E-mail: biere@informatik.uni-freiburg.de

Manuel Kauers
Institute for Algebra, Johannes Kepler University, Linz, Austria
E-mail: manuel.kauers@jku.at

rules too. We demonstrate the different proof formats on the use case of arithmetic circuit verification and discuss how these proofs can be produced as a by-product in formal verification. We present the proof checkers PACHECK, PASTÈQUE, and NUSS-CHECKER. PACHECK checks proofs in practical algebraic calculus more efficiently than PASTÈQUE, but the latter is formally verified using the proof assistant Isabelle/HOL. The tool NUSS-CHECKER is used to check proofs in the Nullstellensatz format.

Keywords Algebraic Proof Systems · Nullstellensatz Proofs · Polynomial Calculus · Gröbner Basis · Arithmetic Circuit Verification · Isabelle/HOL

1 Introduction

Formal verification aims to guarantee the correctness of a given system with respect to a certain specification. However, the verification process might not be error-free and return incorrect results, even in well-known systems such as Mathematica [15]. In order to guarantee the correctness of the outcome, one would have to formally verify the verification tool, e.g., using a theorem prover, which typically is a demanding task and for complex software it is often infeasible. Thus, a more common technique to increase the trust in verification results is to generate proof certificates, which monitor steps of the verification process and enables reproducing the proof. These certificates can be checked by a simple stand-alone proof checker.

For example, many applications of formal verification use satisfiability (SAT) solving and various resolution or clausal proof formats [20], such as DRUP [57, 58], DRAT [22], and LRAT [14] are available to validate the verification results. In the annual SAT competition it is even required to provide certificates since 2013. However, in certain applications SAT solving cannot be applied successfully. For instance formal verification of arithmetic circuits, more precisely of multiplier circuits, is considered to be hard for SAT solving.

Automated reasoning based on computer algebra has a long history [27–29] with renewed recent interest. The general idea of this approach is to reformulate a problem as a question about sets of multivariate polynomials, then do Gröbner bases [8] computations and use properties of Gröbner bases to answer the question.

Formal verification using computer algebra provides one of the state-of-the-art techniques in verifying gate-level multipliers [11, 34, 46, 47]. In this approach the circuit is modeled as a set of polynomials and it is shown that the specification, also encoded as a polynomial, is implied by the polynomials that are induced by the circuit. More precisely, for each logical gate in the circuit a polynomial equation is defined that captures the relations of the inputs and output of the gate. The polynomials are sorted according to a term ordering that is consistent with the topological order of the circuit. This has the effect that these gate polynomials automatically generate a Gröbner basis [8]. Preprocessing techniques based on variable elimination are applied to simplify the representation of the Gröbner basis [34, 46]. After preprocessing the specification polynomial is reduced by the simplified gate polynomials using a multivariate polynomial division with remainder until no further reduction is possible. The given multiplier is correct if and only if the final result is zero.

Furthermore, algebraic reasoning in combination with SAT is successfully used to solve complex combinatorial problems [7], e.g., finding faster ways for matrix multiplication [23,24], computing small unit-distance graphs with chromatic number 5 [19], or solving the Williamson conjecture [6], and has possible future applications in cryptanalysis [10,56]. All these applications raise the need to invoke algebraic proof systems for proof validation.

Two algebraic proof systems are commonly studied in the proof complexity community, polynomial calculus (PC) [12], and Nullstellensatz (NSS) [3]. Both systems allow reasoning over polynomial equations where the variables represent Boolean values. These proof systems are well-studied, with the main focus on deriving complexity measures, such as degree and proof size, e.g., [2,26,49,50].

Proofs in PC allow us to dynamically capture whether a polynomial can be derived from a given set of polynomials using algebraic ideal theory. However, PC as originally defined [12] is not suitable for effective proof checking [31], because information of the origin of the proof steps is missing. We introduce the practical algebraic calculus (PAC) [54], which includes this information and therefore can be checked efficiently. A proof in PAC is a sequence of proof steps, which model single polynomial operations. During proof checking each proof step is checked for correctness. Thus, whenever the proof contains an error, we are able to pinpoint the incorrect proof step.

In the first version of PAC [54] we explicitly require to write down all polynomial equations, including exponents, which leads to very large proof files. Since in our application all variables represent elements of the Boolean domain, we can impose for each variable x the equation $x^2 = x$. We use this observation and specialize PAC to treat exponents implicitly. That is, we immediately reduce all exponents greater than one in the polynomial calculations. Furthermore, we add an indexing scheme to PAC to address polynomial equations and add deletion rules for efficiency. We include a formalization of extension rules that allow us to merge and check combined proofs obtained from SAT and computer algebra [35] in a uniform (and now precise) manner (Sect. 2).

Proofs in NSS capture whether a polynomial can be represented as a linear combination of a given set of polynomials. These proofs are very concise as they consist only of the input polynomials and the sequence of corresponding co-factor polynomials. However, if the resulting polynomial is not equal to the desired target polynomial, it is unclear how to locate the error in the proof. Furthermore, it is impossible to express intermediate optimizations and rewriting techniques on the given set of polynomials in NSS, because we are not able to explicitly model preprocessing steps. We conjectured for the application of multiplier circuit verification [31] that: "In a correct NSS proof we would also need to express the rewritten polynomials as a linear combination of the given set of polynomials and thus loose the optimized representation, which will most likely lead to an exponential blow-up of monomials in the NSS proof." Surprisingly, we have to reject our conjecture, at least for those multiplier architectures that are considered in our approach and our experimental results demonstrate that we are able to generate compact NSS proofs.

In this article we introduce LPAC, a PAC format including linear combinations that combines PAC with the strength of NSS (Sect. 3), namely a shorter proof, while retaining the possibility to identify errors. All proof formats can be produced

by our verification tool AMulet 2.0 [33]. Depending on the options the proofs will have a stronger PAC, a hybrid, or a stronger NSS flavor (Sect. 4).

We present our new proof checkers Pacheck and Pastèque. They support PAC (Sect. 5). The proof checker Pastèque in contrast to Pacheck is verified in Isabelle/HOL, but Pacheck is faster and more memory efficient. To (in)validate our conjecture, we also implemented an NSS checker, Nuss-Checker. This gives us the evidence that NSS proofs do not lead to an exponential blow-up (Sect. 6). Therefore, we also extend Pacheck and Pastèque to check LPAC proofs (Sect. 7).

The tools are easy to use and their results can easily be interpreted. We experiment with the verification of various multipliers that require our new extensions to be checked. The new PAC format makes the proofs easier to check and less memory hungry, but proofs in LPAC achieve even better performance for both checkers (Sect. 8).

This article extends and revises work presented earlier [32, 39, 54]. As a novelty we introduce LPAC, the modification of the PAC format [39] to additionally support linear combinations of polynomials in the proof rules. Hence, we are able to not only simulate NSS and PC proofs in PAC, but we are also able to derive hybrid proofs that consist of a sequence of linear combinations. The hybrid format allows us to generate concise proofs, which are faster to check by our new checkers (Sect. 8), and where errors in the proof can be located. We present how LPAC proofs on different abstraction levels, i.e., NSS, hybrid or PC, are generated in our recent verification tool AMulet 2.0 [33]. Extending [39], we highlight necessary modifications in our proof checkers Pacheck and Pastèque to cover LPAC.

2 Algebraic Proof Systems

In this section we introduce the proof systems polynomial calculus (PC) [12] and its instantiation PAC (Sect. 2.1) and Nullstellensatz [3] (Sect. 2.2). Our algebraic setting follows [13] and we assume $0 \in \mathbb{N}$.

- Let R be a ring and X denote the set of variables $\{x_1, \ldots, x_l\}$. By $R[X]$ we denote the ring of polynomials in variables X with coefficients in R.
- A *term* $\tau = x_1^{d_1} \cdots x_l^{d_l}$ is a product of powers of variables for $d_i \in \mathbb{N}$. A *monomial* is a multiple of a term $c\tau$ with $c \in R \setminus \{0\}$ and a *polynomial* is a finite sum of monomials with pairwise distinct terms.
- On the set of terms $[X]$ an order $\leq$ is fixed such that for all terms τ, σ_1, σ_2 it holds that $1 \leq \tau$ and further $\sigma_1 \leq \sigma_2 \Rightarrow \tau\sigma_1 \leq \tau\sigma_2$. One such order is the so-called *lexicographic term order*, defined as follows. If the variables of a polynomial are ordered $x_1 > x_2 > \cdots > x_l$, then for any two distinct terms $\sigma_1 = x_1^{d_1} \cdots x_l^{d_l}$, $\sigma_2 = x_1^{e_1} \cdots x_l^{e_l}$ we have $\sigma_1 < \sigma_2$ iff there exists an index i with $d_j = e_j$ for all $j < i$, and $d_i < e_i$. We have $\sigma_1 = \sigma_2$ iff $d_j = e_j$ for all $1 \leq j \leq l$.
- For a polynomial $p = c\tau + \cdots$ the largest term τ (w.r.t. $\leq$) is called the *leading term* $\mathrm{lt}(p) = \tau$. The *leading coefficient* $\mathrm{lc}(p) = c$ and *leading monomial* $\mathrm{lm}(p) = c\tau$ are defined accordingly. We call $\mathrm{tail}(p) = p - \mathrm{lm}(p)$ the *tail* of p.

As we will only consider polynomial equations with right hand side zero, we take the freedom to write f instead of $f = 0$. In our setting all variables represent Boolean variables, i.e., we are only interested in solutions where every variable $x \in X$ is assigned either 0 or 1. We can therefore impose the equations $x^2 - x = 0$

for all variables x. The set $B(X) = \{x^2 - x \mid x \in X\} \subset R[X]$ is called the set of Boolean value constraints. Note that R is still an arbitrary ring as we do not restrict the coefficients of the polynomials, we only restrict the values of the variables.

Definition 1 For a set $G \subseteq R[X]$, a *model* is a point $u = (u_1, \ldots, u_l) \in R^l$ such that $\forall g \in G : g(u) = g(u_1, \ldots, u_l) = 0$. Here, by $g(u_1, \ldots, u_l)$ we mean the element of R obtained by evaluating the polynomial g for $x_1 = u_1, \ldots, x_l = u_l$. Given $S \subseteq R$ a set $G \subseteq R[X]$ and a polynomial $f \in R[X]$, we write $G \models_S f$ if every model for G is also a model for $\{f\}$, i.e., $G \models_S f \iff \forall u \in S^l : \forall g \in G : g(u) = 0 \Rightarrow f(u) = 0$.

Algebraic proof systems typically reason about polynomial equations. Given $G \subseteq R[X]$ and $f \in R[X]$, the aim is to show that an equation $f = 0$ is implied by the constraints $g = 0$ for every $g \in G \cup B(X)$. This means that every common Boolean root of the polynomials $g \in G$ is also a root of f. In algebraic terms, we want to derive whether f belongs to the ideal generated by $G \cup B(X)$.

Definition 2 A nonempty subset $I \subseteq R[X]$ is called an *ideal* if $\forall u, v \in I : u + v \in I$ and $\forall w \in R[X], \forall u \in I : wu \in I$. If $G = \{g_1, \ldots, g_m\} \subseteq R[X]$, then the ideal generated by G is defined as $\langle G \rangle = \{q_1 g_1 + \cdots + q_m g_m \mid q_1, \ldots, q_m \in R[X]\}$.

Definition 3 Let $G \subseteq R[X]$ be a finite set of polynomials. A polynomial $f \in R[X]$ can be *deduced* from G if $f \in \langle G \rangle$. In this case we write $G \vdash f$.

2.1 Polynomial Calculus and PAC

The first proof system we consider is PC [12]. We discuss the original definition [12] over fields in Sect. 2.1.1 and generalize the soundness and completeness arguments. In Sect. 2.1.2 we generalize the correctness arguments to commutative rings with unity, when the constraint set G has a certain shape. For completeness the property "commutative ring with unity" is not sufficient and we will require stronger assumptions on the constraint set G in Sect. 2.1.2. In Sect. 2.1.3 we present our instantiation PAC.

2.1.1 Polynomial Calculus over Fields

In the original definition of PC [12] the coefficient ring R is assumed to be a field $\mathbb{K}$. Let $G \subseteq \mathbb{K}[X]$ and $f \in \mathbb{K}[X]$. A proof in PC is a sequence of polynomials $P = (p_1, \ldots, p_m)$ which are deduced by repeated application of the following proof rules:

Addition	$\dfrac{p_i \qquad p_j}{p_i + p_j}$	p_i, p_j appears earlier in the proof or are contained in G
Multiplication	$\dfrac{p_i}{q p_i}$	p_i appears earlier in the proof or is contained in G and $q \in \mathbb{K}[X]$ being arbitrary

We present here a variant of the PC where the addition and multiplication rules are closely related to the definition of an ideal. In the initial definition of PC [12], the addition rule is in fact a linear combination rule and includes multiplication by

constants. The multiplication rule is more restrictive and only allows multiplication by a single variable $x \in X$ [12] or multiplication with any term, e.g., [9] instead of a polynomial. It is easy to see that our definition of PC and the original definition are equivalent and are able to simulate each other polynomially.

Note that every element p_i of a PC proof P is an element of the ideal generated by G. This means that every common root of the elements of G is also a root of every polynomial appearing in the proof.

Thanks to the theory of Gröbner bases [4, 8, 13] the polynomial calculus is decidable, i.e., there is an algorithm which for any finite $G \subseteq \mathbb{K}[X]$ and $f \in \mathbb{K}[X]$ can decide whether $G \vdash f$ or not.

A basis of an ideal I is called a Gröbner basis if it enjoys certain structural properties whose precise definitions are not relevant for our purpose. What matters are the following fundamental facts:

- There is an algorithm (Buchberger's algorithm) which for any given finite set $G \subseteq \mathbb{K}[X]$ computes a Gröbner basis H for the ideal $\langle G \rangle = \langle H \rangle$ generated by G.
- Given a Gröbner basis H, there is a computable function $\mathrm{red}_H \colon \mathbb{K}[X] \to \mathbb{K}[X]$ such that $\forall\, p \in \mathbb{K}[X] : \mathrm{red}_H(p) = 0 \iff p \in \langle H \rangle$.
- Moreover, if $H = \{h_1, \ldots, h_m\}$ is a Gröbner basis of an ideal I and $p, r \in \mathbb{K}[X]$ are such that $\mathrm{red}_H(p) = r$, then there exist $q_1, \ldots, q_m \in \mathbb{K}[X]$ such that $p - r = q_1 h_1 + \cdots + q_m h_m$, and such co-factors q_i can be computed.

In [12] soundness and completeness are shown for degree-bounded polynomials. In this context *soundness* means that every polynomial f which can be deduced by the rules of PC from a given set of polynomials G vanishes on every common root of the polynomials $g \in G$, i.e., $G \vdash f \implies G \models_{\mathbb{K}} f$. *Completeness* means whenever a polynomial f cannot be deduced by the rules of PC from G, then there exists a common root of the polynomials G where f does not evaluate to zero, i.e., $G \nvdash f \implies G \not\models_{\mathbb{K}} f$, or equivalently $G \models_{\mathbb{K}} f \implies G \vdash f$. We are able to generalize these arguments in this article without forcing a bound on the degree of f and the polynomials in G. At the end of this section we summarize how the results fit together in the context of algebraic verification.

To show soundness and completeness of PC over fields $\mathbb{K}$, we now introduce the extended calculus with the additional radical rule [13, Chap. 4§2 Def 2].

$$\text{Radical} \quad \frac{p^m}{p} \quad \begin{array}{l} m \in \mathbb{N} \setminus \{0\} \text{ and} \\ p^m \text{ appears earlier in the proof or is contained in } G. \end{array}$$

Definition 4 If the polynomial f can be deduced from the polynomials in G with the rules of PC and this additional radical rule, we write $G \vdash^+ f$ and call this proof *radical proof*. In algebra, the set $\{\, f \in \mathbb{K}[X] : G \vdash^+ f \,\}$ is called the *radical ideal* of G and is typically denoted by $\sqrt{\langle G \rangle}$.

Theorem 1 *Let $\mathbb{K}$ be an algebraically closed field and $G \subseteq \mathbb{K}[X]$, $f \in \mathbb{K}[X]$. It holds*

$$G \vdash^+ f \iff G \models_{\mathbb{K}} f.$$

Proof It follows from Hilbert's Nullstellensatz [13, Chap. 4§1 Thms. 1 and 2] that the set of all models of G is nonempty if and only if $1 \notin \langle G \rangle$, and furthermore we have $G \vdash^+ f \iff G \models_{\mathbb{K}} f$.

We are able to derive from Thm. 1 that the extended PC including the radical rule is correct ("⇒") and complete ("⇐").

Also the extended calculus $\vdash^+$ is decidable. It can be reduced to $\vdash$ using the so-called Rabinowitsch trick [13, Chap. 4§2 Prop. 8], which says

$$f \in \sqrt{\langle G \rangle} \iff 1 \in \langle G \cup \{yf - 1\} \rangle \qquad \text{or} \qquad G \vdash^+ f \iff G \cup \{yf - 1\} \vdash 1,$$

depending whether you prefer algebraic or logic notation. In both cases, y is a new variable and the ideal/theory on the right hand sides is understood as an ideal/theory of the extended ring $\mathbb{K}[X, y]$.

Corollary 1 *Let $\mathbb{K}$ be an algebraically closed field and assume $G \subseteq \mathbb{K}[X]$, $f \in \mathbb{K}[X]$, and $y \notin X$. We have $G \cup \{yf - 1\} \vdash 1 \iff G \models_{\mathbb{K}} f$.*

The Rabinowitsch trick is therefore used to replace a radical proof ($\vdash^+$) by a PC refutation and we can therefore decide the existence of models and furthermore produce certificates for the non-existence of models using only the basic version of PC. Thus, we do not have to consider the radical rule in practice.

In Thm. 1 we consider models $u \in \mathbb{K}^l$. For our applications, only models $u \in \{0, 1\}^l = \mathbb{B}^l \subseteq \mathbb{K}^l$ matter. Using basic properties of ideals [13, Chap. 4§3 Thm. 4], it is easy to show for $G \subseteq \mathbb{K}[X]$, $f \in \mathbb{K}[X]$ that $G \models_{\mathbb{B}} f \iff G \cup B(X) \models_{\mathbb{K}} f$. Recall from Def. 1 that $G \models_{\mathbb{B}} f \iff \forall u \in \mathbb{B}^l : \forall g \in G : g(u) = 0 \Rightarrow f(u) = 0$.

Furthermore, the equivalence $G \cup B(X) \vdash^+ f \iff G \cup B(X) \models_{\mathbb{K}} f$ holds even when $\mathbb{K}$ is not algebraically closed, because changing from $\mathbb{K}$ to its algebraic closure $\overline{\mathbb{K}}$ will not have any effect on the models in $\mathbb{B}^l$. Finally, let us remark that the finiteness of $\mathbb{B}^l$ also implies that $G \cup B(X) \vdash^+ f \iff G \cup B(X) \vdash f$. This follows from Seidenberg's lemma [4, Lemma 8.13] and generalizes Thm. 1 of [12].

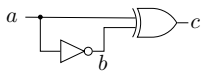
Corollary 2 *Let $G \subseteq \mathbb{K}[X]$, $f \in \mathbb{K}[X]$, for any field $\mathbb{K}$. Then the following holds: $G \cup B(X) \vdash f \iff G \models_{\mathbb{B}} f$.*

Let us briefly put the results of this section into context on the use case of formal verification. In algebraic verification the set G denotes the initial constraint set, e.g., for verifying circuits G contains all polynomials induced by a given circuit. The polynomial f encodes the specification. The goal of verification is to derive, whether f is implied by G, meaning that all common roots of the polynomials in G are roots of f, i.e. $G \models_{\mathbb{K}} f$. From $G \vdash f$ it trivially follows that $G \models_{\mathbb{K}} f$. However, the other direction $G \nvdash f \implies G \not\models_{\mathbb{K}} f$ does not hold in general. From Hilbert's Nullstellensatz, cf. Thm. 1, we are only able to derive that $G \nvdash^+ f \implies G \not\models_{\mathbb{K}} f$.

This means that in general an ideal membership test is not sufficient for verification and we would need to involve the stronger radical membership test to prove non-existence of models. Using the Rabinowitsch trick, cf. Cor. 1, allows us to replace the radical proof by an ideal membership test.

If all variables are Boolean, which is often the case in algebraic verification, we can further simplify Thm. 1, cf. Cor. 2. First, we relax on $\mathbb{K}$ being algebraically closed, because we are only considering a finite number of models $\mathbb{B}^l$. Second, because of the finiteness of $\mathbb{B}^l$, $G \cup B(X)$ is a zero-dimensional ideal, and using Seidenbergs's Lemma we are able to deduce $\langle G \cup B(X) \rangle = \sqrt{\langle G \cup B(X) \rangle}$. Thus, we are able to replace the radical proof in Thm. 1 by an ideal membership test.

$$G \cup B(X) = \{ -b+1-a,\ -c+a+b-2ab,\ a^2-a,\ b^2-b,\ c^2-c\}$$



$$\dfrac{\dfrac{\dfrac{-c+a+b-2ab \qquad -b+1-a}{-c+1-2ab} \qquad \dfrac{-b+1-a}{2ab-2a+2a^2}}{-c+1-2a+2a^2} \qquad \dfrac{a^2-a}{-2a^2+2a}}{-c+1}$$

Fig. 1: The circuit, polynomial representation of the gates and proof for Ex. 1.

Example 1 This example shows that the output c of an XOR gate over an input a and its negation $b = \neg a$ is always true, i.e., $c = 1$ or equivalently $-c + 1 = 0$. We apply the polynomial calculus over the ring $R[X] = \mathbb{K}[X] = \mathbb{Q}[c, b, a]$. Over $\mathbb{Q}$ a NOT gate $x = \neg y$ is modeled by the polynomial $-x + 1 - y$ and an XOR gate $z = x \oplus y$ is modeled by the polynomial $-z + x + y - 2xy$. Because $X = \{a, b, c\}$, we have $B(X) = \{a^2 - a, b^2 - b, c^2 - c\}$. The corresponding circuit representation, the constraint set $G \cup B(X)$, and a polynomial proof tree are shown in Fig. 1.

2.1.2 Polynomial Calculus over commutative rings with unity

For certain sets of polynomials G we are further able to generalize the soundness and completeness arguments for rings R, which not necessarily have to be fields, e.g., $R = \mathbb{Z}$. Let now R denote a commutative ring with unity. By $R^\times$ we denote the set of multiplicatively invertible elements of R. The rules of PC remain unaffected.

Definition 5 Let $G \subseteq R[X]$. If for a certain term order, all leading terms of G only consist of a single variable with exponent 1 and are unique and further $\mathrm{lc}(g) \in R^\times$ for all $g \in G$, then we say G has *unique monic leading terms* (UMLT). Let $X_0(G) \subseteq X$ be the set of all variables that do not occur as leading terms in G.

Example 2 The set $G = \{-x+2y, y-z\} \subseteq \mathbb{Z}[x, y, z]$ has UMLT for the lexicographic term order $x > y > z$. In this case $X_0(G) = \{z\}$.

Definition 6 Let $\varphi\colon X \to \mathbb{B} \subseteq R$ denote an *assignment* of all variables X. We extend φ to an evaluation of polynomials in the natural way, i.e., $\varphi\colon R[X] \to R$.

Theorem 2 (Soundness) *Let $G \subseteq R[X]$ be a finite set of polynomials and $f \in R[X]$, then*

$$G \cup B(X) \vdash f \quad \Rightarrow \quad G \models_{\mathbb{B}} f.$$

Proof If $G \cup B(X) \vdash f$ then $f \in \langle G\rangle + \langle B(X)\rangle$ by definition. This means there are $u_1, \ldots, u_m \in R[X]$ and $v_1, \ldots, v_r \in R[X]$ with $f = u_1 g_1 + \cdots + u_m g_m + v_1 b_1 + \cdots + v_r b_r$, where $g_i \in G$ and $b_i = x_i(x_i - 1) \in B(X)$ for $i = 1 \ldots r$. Any assignment φ in the sense of Def. 6 vanishes on $B(X)$, i.e., $\varphi(b_i) = 0$. If φ is also a model of G then $\varphi(g_i) = 0$ too and as a consequence $\varphi(f) = 0$. Therefore $G \models_{\mathbb{B}} f$, as claimed.

Completeness is less obvious. Consider for instance that $\{2x\} \models_{\mathbb{B}} x$ but $x \notin \langle 2x \rangle$ in $\mathbb{Z}[X]$. Requiring G to have UMLT turns out to be essential (which $\{2x\}$ does not have in $\mathbb{Z}[X]$, because $2 \notin \mathbb{Z}^{\times}$). Additionally, we will require the considered ring R to be an *integral domain*, which satisfies the property that the product of any two nonzero elements is nonzero [13].

Lemma 1 *If $G \models_{\mathbb{B}} p$ and $G \models_{\mathbb{B}} q$ then $G \models_{\mathbb{B}} q \pm p$.*

Lemma 2 *Let $G \subseteq R[X]$ be a finite set of polynomials with UMLT. Then for all $q \in R[X]$ there exist $p \in \langle G \rangle + \langle B(X) \rangle$ and $r \in R[X_0(G)]$ with $q = p + r$, such that the variables in the monomials in r have only exponents 1.*

Proof We construct p and r by division of q by the polynomials in $G \cup B(X)$ until no term in r is divisible by any leading term of $G \cup B(X)$. First, we reduce q by the polynomials of G. Let $g_1 \in G$. Using polynomial division we are able to calculate $f_1, r_1 \in R[X]$ such that $q = f_1 g_1 + r_1$ and no term in r_1 is a multiple of the leading term of g_1. We continuously divide the remainder by polynomials of G and derive $q = f_1 g_1 + \cdots + f_m g_m + r_m$ for $g_i \in G$, $f_i, r_m \in R[X]$.

This process has to terminate because the tail of a polynomial contains only smaller variables and the number of variables in G is finite. Since G has UMLT, r_m contains only variables in $X_0(G)$ which do not occur as leading terms, i.e, $r_m \in R[X_0(G)]$. If any of these variables occurs with exponent larger than one we can use $B(X)$ to reduce their exponent to 1. Hence, we are able to derive $q = f_1 g_1 + \cdots + f_m g_m + v_1 b_1 + \cdots + v_l b_l + r$, where $g_i \in G$, $b_i \in B(X)$, and $f_i, v_i \in R[X]$ and define $p = f_1 g_1 + \cdots + f_m g_m + v_1 b_1 + \cdots + v_l b_l$.

Example 3 Let $G \subseteq \mathbb{Z}[x, y, z]$ be as in Ex. 2 and assume $q = 2x^2 + xy + z^2 \in \mathbb{Z}[x, y, z]$. Consequently

$$\begin{aligned} p &= (-2x - 5y)(-x + 2y) + (10y + 10z)(y - z) - 11(-z^2 + z) \\ &= 2x^2 + xy + z^2 - 11z \ \in \ \langle G \rangle + \langle B(X) \rangle \quad \text{and} \end{aligned}$$

$r = 11z \ \in \ \mathbb{Z}[X_0(G)]$.

Lemma 3 *Assume that R is an integral domain. Let $p \in R[X]$ with $p^2 - p \in \langle B(X) \rangle = \langle \{x^2 - x \mid x \in X\} \rangle$. Further let φ be an assignment in the sense of Def. 6. Then $\varphi(p) \in \mathbb{B} = \{0, 1\}$.*

Proof Since $p^2 - p \in \langle B(X) \rangle$ there are $f_i \in R[X]$ with $p^2 - p = \sum_i f_i \cdot (x_i^2 - x_i)$. Thus, $\varphi(p^2 - p) = 0$, as φ vanishes on $B(X)$. Assume now $\varphi(p) = \epsilon$ with $\epsilon \in R$. Then $\varphi(p^2 - p) = \varphi(p)^2 - \varphi(p) = \epsilon^2 - \epsilon = \epsilon(\epsilon - 1)$. As R is an integral domain, only $\epsilon \in \mathbb{B}$ yields $\varphi(p^2 - p) = 0$.

Theorem 3 (Completeness) *Let R be an integral domain and let $G \subseteq R[X]$ be a finite set of polynomials with UMLT. Suppose further that*

$$\forall g \in G : (\mathrm{lc}(g)^{-1} \, \mathrm{tail}(g))^2 + (\mathrm{lc}(g)^{-1} \, \mathrm{tail}(g)) \in \langle B(X) \rangle.$$

Then for every $f \in R[X]$ we have

$$G \models_{\mathbb{B}} f \quad \Rightarrow \quad G \cup B(X) \vdash f.$$

Proof Suppose we have $G \models_{\mathbb{B}} f$. Then our goal is to show $f \in \langle G\rangle + \langle B(X)\rangle$. First, by applying Lemma 2, we obtain $p \in \langle G\rangle + \langle B(X)\rangle$ and $r \in R[X_0(G)]$ with $f = p + r$. Thus $G \cup B(X) \vdash p$ by definition. Using Thm. 2 we derive $G \models_{\mathbb{B}} p$ and accordingly $G \models_{\mathbb{B}} f - p = r$ by Lemma 1. Now assume $r \neq 0$ and let m be a monomial of r which contains the smallest number of variables. Consider the assignment φ that maps $x \in X_0(G)$ to 1 if it appears in m and to 0 otherwise. Therefore $\varphi(r) \neq 0$ since the coefficient of m is unequal to 0. This assignment on $X_0(G)$ admits a unique extension to X which vanishes on G. First, we consider the polynomial $\alpha x + t \in G$, where $\alpha \in R^\times$ and $t = \operatorname{tail}(g)$, with the smallest leading term x. For this polynomial all variables in t are already considered in φ. Since $\alpha x + t = 0 \Leftrightarrow x = -\alpha^{-1}t$ and we require $(\operatorname{lc}(g)^{-1}\operatorname{tail}(g))^2 + (\operatorname{lc}(g)^{-1}\operatorname{tail}(g)) = (\alpha^{-1}t)^2 + (\alpha^{-1}t) = (-\alpha^{-1}t)^2 - (-\alpha^{-1}t) \in \langle B(X)\rangle$, we have $\varphi(-\alpha^{-1}t) \in \{0,1\}$ by Lemma 3. We extend the assignment φ to x by choosing $\varphi(x) = \varphi(-\alpha^{-1}t)$. We continue in this fashion until all leading terms of G are assigned. Since G has UMLT we are able to derive such an assignment φ, which contradicts $G \models_{\mathbb{B}} r$. Thus $r = 0$ and $f = p + r \in \langle G\rangle + \langle B(X)\rangle$.

In an earlier version of the manuscript, as well as in the conference paper [34, Thm. 2], the assumptions "$\forall g \in G : (\operatorname{lc}(g)^{-1}\operatorname{tail}(g))^2 + (\operatorname{lc}(g)^{-1}\operatorname{tail}(g)) \in \langle B(X)\rangle$" and "$R$ is an integral domain" were missing. We thank one of the referees for making us aware of these bugs. If any of the three assumptions of Thm. 3 is missing, the theorem is wrong, as can be seen in the following examples.

First, let $G = \{xyz + xy - x - y\} \subseteq \mathbb{Z}[x,y,z]$ and $f = x - y \in \mathbb{Z}[x,y,z]$. The ring $R = \mathbb{Z}$ is an integral domain and we have $(xy-x-y)^2+xy-x-y \in \langle B(X)\rangle$. However G does not have UMLT, because the leading term of $xyz + xy - x - y$ consists of more than one variable. We have $G \models_{\mathbb{B}} f$ with the models $(x,y,z) = (0,0,0)$, $(0,0,1)$, and $(1,1,1)$, but $G \cup B(X) \not\vdash f$ because $r = x - y$.

Next, consider $G = \{-x+2y\} \subseteq \mathbb{Z}[x,y]$ and $f = y \in \mathbb{Z}[x,y]$. The polynomials in G have UMLT and $\mathbb{Z}$ is an integral domain. However, for the polynomial $-x + 2y$ we have $4y^2 - 2y \notin \langle B(X)\rangle$. We have $G \models_{\mathbb{B}} f$ with the model $(x,y) = (0,0)$ but $G \cup B(X) \not\vdash f$ because $r = y$.

Finally, let $G = \{x + 4y\} \subseteq \mathbb{Z}_{10}[x,y]$ and $f = y \in \mathbb{Z}_{10}[x,y]$. The polynomial in G has UMLT, and we have $(4y)^2 + 4y = 6y^2 - 6y \in \langle B(X)\rangle$. However the ring $R = \mathbb{Z}_{10}$ is not an integral domain as $5 \cdot 2 = 0$. We have $G \models_{\mathbb{B}} f$ with the model $(x,y) = (0,0)$, but $G \cup B(X) \not\vdash f$ because $r = y$.

Although the previous example shows that the assumption that R is an integral domain cannot simply be dropped from Thm. 3, it is somewhat stronger than necessary. What really enters through Lemma 3 into the proof of Thm. 3 is the assumption that R is a ring in which the formula $\forall\, x \in R : x(x-1) = 0 \Rightarrow x = 0 \vee x = 1$ is true. This holds in every integral domain, but also in some rings that are not integral domains, for example in rings $\mathbb{Z}_{2^k}$ for $k > 1$. In our use case of algebraic circuit verification, which we introduce in Sect. 4.1, we choose $R = \mathbb{Z}_{2^k}$ for $k \geq 1$ to admit modular reasoning [34]. In the following lemma, we use Hensel lifting to prove that the rings $\mathbb{Z}_{2^k}$ have the desired property.

Lemma 4 *Let $k \in \mathbb{N} \setminus \{0\}$, let φ be an assignment in the sense of Def. 6, and let $p \in \mathbb{Z}_{2^k}[X]$ be such that $p^2 - p \in \langle B(X)\rangle$. Then $\varphi(p) \in \mathbb{B} = \{0,1\}$.*

Proof Proof by induction over k. Base case $k = 1$: For $k = 1$ the ring $\mathbb{Z}_2$ is a field. Since every field is an integral domain the base case follows by Lemma 3.

Induction step $k \to k+1$: Assume $p \in \mathbb{Z}_{2^{k+1}}[X]$ with $\varphi(p^2 - p) = 0 \mod 2^{k+1}$. Let now $\varphi(p) = \epsilon$ with $\epsilon \in \mathbb{Z}_{2^{k+1}}$. Since $\epsilon \in \{0, \ldots, 2^{k+1} - 1\}$ we can write $\epsilon = 2^k\epsilon_1 + \epsilon_0$ for $\epsilon_1 \in \{0, 1\}$, $\epsilon_0 \in \{0, \ldots, 2^k - 1\}$:

$$\begin{aligned} & \varphi(p^2 - p) = \varphi(p)^2 - \varphi(p) = \epsilon(\epsilon - 1) = 0 \mod 2^{k+1} \\ \Longrightarrow \quad & (2^k\epsilon_1 + \epsilon_0)(2^k\epsilon_1 + \epsilon_0 - 1) = 0 \mod 2^{k+1} \\ \Longrightarrow \quad & 2^{2k}\epsilon_1^2 + 2^k\epsilon_1(\epsilon_0 - 1) + 2^k\epsilon_1\epsilon_0 + \epsilon_0(\epsilon_0 - 1) = 0 \mod 2^{k+1} \end{aligned}$$

First, since $k \geq 1$, we have $2^{2k} = 0 \mod 2^{k+1}$. Second, it follows that $\epsilon_0(\epsilon_0 - 1) = 0 \mod 2^k$. Thus by the induction hypothesis we have $\epsilon_0 \in \{0, 1\}$ and the equation above simplifies to

$$\begin{aligned} & 2^{2k}\epsilon_1^2 + 2^k\epsilon_1(\epsilon_0 - 1) + 2^k\epsilon_1\epsilon_0 + \epsilon_0(\epsilon_0 - 1) = 0 \mod 2^{k+1} \\ \Longrightarrow \quad & 2^k\epsilon_1(\epsilon_0 - 1) + 2^k\epsilon_1\epsilon_0 = 0 \mod 2^{k+1} \\ \Longrightarrow \quad & \epsilon_1(\epsilon_0 - 1) + \epsilon_1\epsilon_0 = 2\epsilon_1\epsilon_0 - \epsilon_1 = \epsilon_1 = 0 \mod 2 \\ \Longrightarrow \quad & \epsilon_1 = 0 \end{aligned}$$

Hence $\varphi(p) = \epsilon = \epsilon_0 \in \{0, 1\}$. □

Corollary 3 (Completeness for $\mathbb{Z}_{2^k}$) *Let $R = \mathbb{Z}_{2^k}$ for $k \geq 1$ and let $G \subseteq R[X]$ be a finite set of polynomials with UMLT. Suppose further that*

$$\forall g \in G : (\mathrm{lc}(g)^{-1}\,\mathrm{tail}(g))^2 + (\mathrm{lc}(g)^{-1}\,\mathrm{tail}(g)) \in \langle B(X)\rangle.$$

Then for every $f \in R[X]$ we have $G \models_{\mathbb{B}} f \Rightarrow G \cup B(X) \vdash f$.

In the use case of algebraic circuit verification, cf. Sect. 4.1, we automatically have "$\forall g \in G : (\mathrm{lc}(g)^{-1}\,\mathrm{tail}(g))^2 + (\mathrm{lc}(g)^{-1}\,\mathrm{tail}(g)) \in \langle B(X)\rangle$". All polynomials $g \in G$ have the form $g := -\mathrm{lt}(g) + \mathrm{tail}(g)$, with $\mathrm{lc}(g) = -1$, and encode the relation between the output and inputs of a gate. The leading term $\mathrm{lt}(g)$ represents the gate output and $\mathrm{tail}(g)$ computes the output signal in terms of the inputs, cf., Fig. 3. Thus $\varphi(\mathrm{tail}(g)) \in \{0, 1\}$ and hence the assumption $\mathrm{tail}(g)^2 - \mathrm{tail}(g) \in \langle B(X)\rangle$ holds.

2.1.3 Practical Algebraic Calculus

PC proofs as defined so far cannot be checked efficiently, because they only contain the conclusion polynomials of each proof step.

Example 4 Consider again the example of Fig. 1. The corresponding PC proof is $P = (-c+1-2ab, 2ab-2a+2a^2, -c+1-2a+2a^2, -2a^2+2a, -c+1)$. To check the correctness of this proof we would need to verify that each polynomial is derived using one of the PC rules, which is hard, because we do not have information on the antecedents.

For practical proof checking we translate the abstract rules of PC into a concrete proof format, i.e., we define a format based on PC, which is logically equivalent but more detailed. In principle a proof in PC can be seen as a finite sequence of polynomials derived from the initial constraint set and previously inferred polynomials by applying either an addition or multiplication rule. To ensure correctness of each proof step it is of course necessary to know which rule was used, to check

Algorithm 1: Proof-Checking$(G \cup B(X), R, f)$

Input : Constraint set $G \cup B(X)$, PAC steps $R = r_1, \ldots, r_k$, target polynomial f
Output: "incorrect" or "correct"

$P_0 \leftarrow G \cup B(X)$;
for $i \leftarrow 1 \ldots k$ **do**
  let $r_i = (o_i, v_i, w_i, p_i)$;
  case $o_i = +$ **do**
    if $v_i \in P_{i-1} \ \wedge \ w_i \in P_{i-1} \ \wedge \ p_i = v_i + w_i$ **then** $P_i \leftarrow \text{append}(P_{i-1}, p_i)$;
    else return "incorrect";
  case $o_i = *$ **do**
    if $v_i \in P_{i-1} \ \wedge \ p_i = v_i * w_i$ **then** $P_i \leftarrow \text{append}(P_{i-1}, p_i)$;
    else return "incorrect";
if $\exists p_i \in P_k \wedge p_i = f$ **then return** "correct" **else return** "incorrect";

that it was applied correctly, and in particular which given or previously derived polynomials are involved. During proof generation these polynomials are usually known and thus we require that all of this information is part of a rule in our concrete PAC proof format to simplify proof checking. A proof rule contains four components

$$o : v, w, p;$$

The first component o denotes the operator which is either '`+`' for addition or '`*`' for multiplication. The next two components v, w specify the two (antecedent) polynomials used to derive p (conclusion). In the multiplication rule w plays the role of the polynomial q of the multiplication rule of PC.

For proof validation we need to make sure that two properties hold. The *connection property* states that the components v, w are either elements of the constraint set or conclusions of previously applied proof rules. For multiplication we only have to check this property for v, because w is an arbitrary polynomial. By the second property, called *inference property*, we verify the correctness of each proof step, namely we simply calculate $v + w$ resp. $v * w$ and check that the obtained result matches p. In a correct PAC proof we further need to verify that at least one conclusion polynomial p matches the target polynomial f. The complete checking algorithm is shown in Alg. 1. Checking each step allows pinpointing the first error, instead of claiming that the proof is wrong somewhere in one of the (usually millions) steps.

Example 5 Consider again the example presented in Ex. 1. One PAC proof obtaining $-c + 1 \in \langle G \cup B(X) \rangle \subseteq \mathbb{Q}[X]$ is:

```
Constraint Set      Proof
-b+1-a;             + : -c+a+b-2a*b,  -b+1-a,        -c+1-2a*b;
-c+a+b-2a*b;        * : -b+1-a,       -2a,           2a*b-2a+2a^2;
a^2-a;              + : -c+1-2a*b,    2a*b-2a+2a^2,  -c+1-2a+2a^2;
b^2-b;              * : a^2-a,        -2,            -2a^2+2a;
c^2-c;              + : -c+1-2a+2a^2, -2a^2+2a,      -c+1;
```

Adaptions We adapt PAC to admit shorter and more concise proofs. First, we index polynomials, i.e., each given polynomial and proof step is labeled by a unique positive number. It can be seen in Ex. 5 that the conclusion polynomial of the first

proof step is again explicitly given as the first antecedent in the third proof step. Using indices, similar to LRAT [14], allows us now to label the first proof step and use this index in the third proof step. Naming polynomials by indices reduces the size of the proof files significantly and makes parsing more efficient, because only the conclusion polynomials of each step and the initial polynomials of G are stated explicitly. However, introducing indices for polynomials has the effect that the semantics changes from sets to multisets, as in DRAT [58], and it is possible to introduce the same polynomial under different names.

Second, we treat exponents implicitly. For bit-level verification [54] only models of the Boolean domain $\{0,1\}^n$ are of interest. Initially, we added the set of Boolean value constraints $B(X) = \{x^2 - x \mid x \in X\}$ to G and have to include steps in the proofs that operate on these Boolean value constraints. Instead, we now handle operations on Boolean value constraints implicitly to reduce the number of proof steps. That is, we remove the Boolean value constraints from the constraint set and when checking the correctness, we immediately reduce exponents greater than one in the polynomials, i.e., $x^2 = x$.

Third, we further introduce a deletion rule to reduce the memory usage of the proof checker. After each proof step the conclusion polynomial will be added to the constraint set, thus the number of stored polynomial increases. If we know that a certain polynomial is not needed anymore in the proof, we use the deletion rule to remove polynomials.

We introduce the semantics of PAC as a transition system. Let P denote a sequence of polynomials which can be accessed via indices. We write $P(i) = \bot$ to denote that the sequence P at index i does not contain a polynomial, and $P(i \mapsto p)$ to denote that P at index i is set to p. The immediate reduction of exponents is denoted by "$\mathrm{mod}\langle B(X)\rangle$". The initial state is $(X = \mathrm{Var}\,(G \cup \{f\}), P)$ where P maps indices to polynomials of G. The following two rules implement the properties of ideals as introduced above for the original PAC.

$$[\textsc{Add}\ (i,j,k,p)] \qquad (X,P) \implies (X, P(i \mapsto p))$$

provided that $P(j) \neq \bot$, $P(k) \neq \bot$, $P(i) = \bot$, $p \in R[X]$, and $p = (P(j) + P(k)) \,\mathrm{mod}\langle B(X)\rangle$.

$$[\textsc{Mult}\ (i,j,q,p)] \qquad (X,P) \implies (X, P(i \mapsto p))$$

provided $P(j) \neq \bot$, $P(i) = \bot$, $p, q \in R[X]$, and $p = (q \cdot P(j)) \,\mathrm{mod}\langle B(X)\rangle$.

In the deletion rule we remove polynomials from P which are not needed anymore in subsequent steps to reduce the memory usage of our tools.

$$[\textsc{Deletion}\ (i)] \qquad (X,P) \implies (X, P(i \mapsto \bot))$$

Example 6 The proof of Ex. 1 in the adapted PAC format. We do not include all possible deletion steps in the proof.

```
Constraint Set              Proof
1 -b+1-a;                   3 +  2,   1,   -c+1-2a*b;
2 -c+a+b-2a*b;              2 d;
                            4 *  1, -2a,        2a*b;
                            1 d;
                            5 +  3,   4,        -c+1;
```

Extension Similar to the *polynomial calculus with resolution* (PCR) [1], which extends PC by a negation rule, we include an extension rule which allows us to add new polynomials to the constraint set. The negation rule of PCR introduces for each variable $x \in X$ an additional variable $\overline{x}$ that represents the negation of x. We generalize this extension rule such that new variables can act as placeholders for polynomials.

We use the extension rule to combine SAT solving and algebraic reasoning in our previous work [34] for multiplier verification. Thus, two proof certificates in different proof systems, DRUP and PAC are generated. In order to derive a single proof certificate we converted DRUP proofs to the PAC format [35]. However, to efficiently convert the resolution steps we encountered the need to extend the initial set of polynomials G to reduce the size of the polynomials (number of monomials) in the PAC proof. We included polynomials of the form $-f_x + 1 - x$, similar to the negation rule in PCR, which introduced the variable f_x as the negation of the Boolean variable x. An example for modelling a resolution step in PAC is given in Ex. 7 below, where the proof step with index 3 demonstrates our new extension rule.

However, at that point we did not apply a proper extension rule, but simply added these extension polynomials to G. This may affect the models of the constraint set, because any arbitrary polynomial can be added as an initial constraint. For example, we could simply add the constant polynomial 1 to G which makes any PAC proof obsolete. To prevent this issue we add an extension rule to PAC, which allows us to add further polynomials to the knowledge base with new variables while preserving the original models on the original variable set of variables X.

$$[\textsc{Ext}\ (i, v, p)] \quad (X, P) \implies (X \cup \{v\}, P(i \mapsto -v + p))$$
$$\text{provided that } P(i) = \bot \text{ and } v \notin X \text{ and } p \in R[X], \text{ and } p^2 - p \in \langle B(X)\rangle.$$

With this extension rule, variables v can act as placeholders for polynomials p, i.e., $-v + p = 0$, which enables more concise proofs. The variables v are not allowed to occur earlier in the proof. Furthermore, to preserve Boolean models, we require $p^2 - p \in \langle B(X)\rangle$. This can be easily checked by calculating $p^2 - p$ and reducing all exponents larger than one to one. The normalized result has to be zero. Without this condition v might take non-Boolean solutions. In that case v^n cannot be simplified to v, requiring to manipulate exponents in the proof checkers, which is currently not supported.

Consider for example $P = \{-y + x - 1\}$. The only Boolean model is $(x, y) = (1, 0)$. If we extend G by $-v + x + 1$ we derive $v = 2$, because $x = 1$ for all models of G. Thus $v^2 - v = 0$ does not hold.

Proposition 1 Ext *preserves the original models on* X.

Proof We show that adding $p_v := -v + p$ does not affect the models of $G \cup B(X) \subseteq R[X]$. We have $\langle G \cup \{p_v\} \cup B(X \cup \{v\})\rangle = \langle G \cup \{p_v\} \cup B(X)\rangle$ because $v^2 - v = p^2 - p$ and $p^2 - p \in \langle B(X)\rangle$. However, every model of $\langle G \cup \{p_v\} \cup B(X)\rangle$ is also a model of $\langle G \cup B(X)\rangle$ because the variable v appears only as leading term in p_v. Hence the result. □

The Isabelle formal proof is very similar to the idea given here, but we have to be more explicit. In particular, we explicitly manipulate a linear combination

of the polynomials and show that every dependence in v can be removed from the linear combination, since the variable v appears only in p_v.

Example 7 Let $\bar{x} \vee \bar{y}$ and $y \vee z$ be two clauses. From these clauses we derive the clause $\bar{x} \vee z$ using resolution. The clauses are translated into polynomial equations using De Morgan's laws and using the fact that a logical AND can be represented by multiplication. For example, from $\bar{x} \vee \bar{y} = \top \Leftrightarrow x \wedge y = \bot$ we derive the polynomial equation $xy = 0$.

For the PAC proof we introduce an extension variable f_z, which models the negation of z, i.e. $-f_z + 1 - z = 0$ in order to find a shorter representation of the second constraint, cf. proof step 5.

```
Constraint Set          Proof
 1 x*y;                  3  = fz, -z+1;
 2 y*z-y-z+1;            4  *  3,  y-1, -fz*y+fz-y*z+y+z-1;
                         5  +  2,    4, -fz*y+fz;
Target                   6  *  1,   fz, fz*x*y;
 -x*z+x;                 7  *  5,    x, -fz*x*y+fz*x;
                         8  +  6,    7, fz*x;
                         9  *  3,    x, -fz*x-x*z+x;
                        10  +  8,    9, -x*z+x;
```

2.2 Nullstellensatz

The *Nullstellensatz* (NSS) *proof system* [3] derives whether a polynomial $f \in R[X]$ can be represented as a linear combination of polynomials from a given set $G = \{g_1, \ldots, g_m\} \subseteq R[X]$. That is, an NSS proof for a given polynomial f and a set $G = \{g_1, \ldots, g_m\}$ is a tuple $P = (h_1, \ldots, h_m)$ of polynomials such that

$$\sum_{i=1}^{m} h_i g_i = f.$$

By the same arguments given for PAC, the soundness and completeness arguments of NSS proofs can be generalized to rings $R[X]$ when G has UMLT. In NSS the Boolean value constraints are treated implicitly to yield shorter proofs. Thus, the NSS proof we consider for a given polynomial $f \in R[X]$ and a set of polynomials $G = \{g_1, \ldots, g_m\} \subseteq R[X]$ is a tuple of co-factors $P = (h_1, \ldots, h_m)$ of polynomials such that there exist polynomials $r_1, \ldots, r_l \in R[X]$ with

$$\sum_{i=1}^{m} h_i g_i + \sum_{i=1}^{l} r_i (x_i^2 - x_i) = f. \tag{1}$$

Checking NSS proofs seems straightforward as we simply need to expand the products $h_i g_i$, calculate the sum, and compare the derived polynomial to the given target polynomial f. However, we discuss practical issues of proof checking in Sect. 6, where we introduce our NSS proof checker Nuss-Checker. Unlike PAC introduced above, NSS does not support extensions.

Example 8 A NSS proof for our running example introduced in Ex. 1 is

```
Constraint Set          Proof
-b+1-a;                 1-2a;
-c+a+b-2a*b;            1;
```

```
letter      ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
number      ::= '0' | '1' | ... | '9'
constant    ::= (number)+
variable    ::= letter (letter | number)*
term        ::= variable ('*' variable)*
monomial    ::= constant | [constant '*'] term
poly        ::= ['-'] monomial ('+' | '-' monomial)*
id          ::= constant
input       ::= (id poly ';')*
lin_com_rule ::= id '%' id ['*' '(' poly ')'] ( '+' id ['*' '(' poly ')']*',' poly ';'
del_rule    ::= id 'd' ';'
ext_rule    ::= id '=' variable ',' poly ';'
proof       ::= (lin_com_rule | del_rule | ext_rule)*
target      ::= poly ';'
```

Fig. 2: Syntax of input polynomials, target, and proofs in the LPAC-format

We derive $(1-2a)(-b+1-a)+(1)(-c+a+b-2ab) = -c+1 \mod \langle B(X)\rangle$ in $\mathbb{Q}[X]$.

3 Merging NSS and PAC into the hybrid proof system LPAC

PAC proofs are very fine-grained, because for each polynomial operation on the constraint set a single proof step is generated and checked for correctness. This makes it on the one hand simple to locate an error in the proof and thus to trace back the error in the automated reasoning tool. On the other hand the proof files are very large as for each proof step we write down a single line consisting of an index, the operation, two antecedents and the conclusion polynomial.

Nullstellensatz proofs are concise, as the core proof only consists of the ordered sequence of the co-factors, which has equal length of the constraint set. Thus the corresponding proof files are typically orders of magnitude smaller than PAC proofs, e.g., compare the proofs in Exs. 6 and 8. However, because proof checking an NSS proof consists of calculating the linear combination and comparing it to the target polynomial, it is impossible to locate a possible error in the proof. Furthermore, the extensions of PAC are not directly portable to core NSS proofs.

To take the best of both worlds we propose now a modified proof format, called LPAC (practical algebraic calculus + linear combinations). It includes a rule to merge the addition and multiplication rule to a single proof rule, which represents linear combination of polynomials. The syntax is given in Fig. 2. Thus we gain the following semantics. Let P denote a sequence of polynomials, which can be accessed via indices. The initial state is $(X = \mathrm{Var}(G \cup \{f\}), P)$ where P maps indices to polynomials of G.

$[\textsc{LinComb}\ (i,(j_1,\ldots,j_n),(q_1,\ldots,q_n),p)] \qquad (X,P) \implies (X, P(i \mapsto p))$

provided that $P(j_1) \neq \bot, \ldots, P(j_n) \neq \bot$, $P(i) = \bot$, $p, q_1, \ldots, q_n \in R[X]$, $n \geq 1$, and $p = (q_1 \cdot P(j_1) + \ldots + q_n \cdot P(j_n)) \bmod \langle B(X)\rangle$.

$[\textsc{Deletion}\ (i)] \qquad (X,P) \implies (X, P(i \mapsto \bot))$

$[\textsc{Ext}\ (i,v,p)] \qquad (X,P) \implies (X \cup \{v\}, P(i \mapsto -v+p))$

provided that $P(i) = \bot$ and $v \notin X$ and $p \in R[X]$, and $p^2 - p \in \langle B(X) \rangle$.

Our new LPAC format allows us to simulate both the PAC format and NSS proofs as follows. The LinComb is able to simulate both the Add and the Mult rule of PAC. By taking $n = 2$, (p_1, p_2), and $(1, 1)$, we obtain the normal Add rule. By taking $n = 1$, (p_1), and (q_1), we obtain Mult. The rules Deletion and Ext remain the same as for PAC. In the actual proof file, elements of the sequence $(q_1, \ldots, q_n)$ can be skipped and are interpreted as the constant sequence 1. We simulate NSS proofs by providing a single LinComb rule in the proof file.

Furthermore, we are able to generate hybrid proofs, which are not as concise as a single linear combination, but also not as fine-grained as an extended PAC proof. For example, in multiplier verification we apply polynomial reductions which always consist of a multiplication and addition of polynomials. In the LPAC proof format we are able to combine these two operations in a single proof step.

Example 9 A possible proof in LPAC for Ex. 1 is as follows:

```
Constraint Set          Proof
1 -b+1-a;               3 %  (1-2a)*1+2, -c+1;
2 -c+a+b-2a*b;          1 d;
```

4 Proof Generation

In this section we demonstrate on the real-world application of multiplier verification how PAC, LPAC, and NSS proofs can be generated. We first provide a brief introduction to multiplier verification using our tool AMulet 2.0, before discussing how proof certificates can be generated.

4.1 Multiplier Verification

We developed a verification tool, called AMulet 2.0 [33,34], which takes as input signed or unsigned integer multipliers C, given as And-Inverter-Graphs (AIGs), with $2n$ input bits $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1} \in \{0, 1\}$ and output bits $s_0, \ldots, s_{2n-1} \in \{0, 1\}$. Nodes in the AIG represent logical conjunction and markings on the edges represent negation. We denote the internal AIG nodes by $l_1, \ldots, l_k \in \{0, 1\}$. Let $\mathbb{Z}[X] = \mathbb{Z}[a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, l_1, \ldots, l_k, s_0, \ldots, s_{2n-1}]$. In our application we require the coefficient domain to be $\mathbb{Z}$, because this allows us to apply modular reasoning by adding a constant 2^k to the set of ideal generators, which helps to keep the size of the intermediate verification results reasonably small. More details on modular reasoning are given in [34].

The multiplier C is correct iff for all possible inputs a_i, $b_i \in \{0, 1\}$ the specification $\mathcal{L} = 0$ holds:

$$\mathcal{L} = -\sum_{i=0}^{2n-1} 2^i s_i + \left(\sum_{i=0}^{n-1} 2^i a_i\right)\left(\sum_{i=0}^{n-1} 2^i b_i\right) \tag{2}$$

The semantics of each AIG node implies a polynomial relation, cf., Fig. 3. Let $G(C) \subseteq \mathbb{Z}[X]$ be the set of polynomials that contains for each AIG node of C the corresponding polynomial relation.

Algorithm 2: Reduction(p, p_v, v)

Input : Polynomials p, $p_v \in \mathbb{Z}[X]$, $\mathrm{lm}(p_v) = -v$
Output: Polynomials h, $r \in \mathbb{Z}[X]$ such that $p + hp_v = r$

```
1 t ← p, r ← p, h ← 0;
2 while t ≠ 0 do
3     if v ∈ lt(t) then
4         h = h + lm(t)/v;
5         r = r + p_v lm(t)/v  mod ⟨B(X)⟩;
6     t = t − lm(t);
7 return h, r
```

The polynomials in $G(C) \cup B(X)$ are ordered according to a lexicographic order, such that the output variable of a gate is always greater than the inputs of the gate, also called *reverse topological term order* (RTTO) [44]. Using this variable ordering leads to $G(C)$ having UMLT.

Let $J(C) = \langle G(C) \cup B(X) \rangle \subseteq \mathbb{Z}[X]$ be the ideal generated by $G(C) \cup B(X)$. The circuit fulfills its specification if and only if we can derive that $\mathcal{L} \in J(C)$, which can be established by reducing $\mathcal{L}$ by the polynomials $G(C) \cup B(X)$ and checking whether the result is zero [34]. The algorithm for reducing a polynomial p by a second polynomial p_v is shown in Alg. 2. We again treat $B(X)$ implicitly, thus we never explicitly reduce by a polynomial from $B(X)$, but always cancel exponents greater than one to one, which is included in line 5. As a reduction order we follow the same order that is established for the variables.

However, simply reducing the specification by $G(C)$ leads to large intermediate results [45]. Hence, we eliminate variables in $G(C)$ prior to reduction to yield a more compact polynomial representation of the circuit [34]. In the preprocessing step, we repeatedly eliminate selected variables $v \in X \setminus X_0$ from $G(C)$, cf. Sect. 4.2. in [36]. Let $p_v \in G(C)$ such that $\mathrm{lt}(p_v) = v$. Since $G(C)$ has UMLT and $v \notin X_0$, such a p_v exists. All polynomials p, with $v \in \mathrm{tail}(p)$ are reduced by p_v to remove v from G using Alg. 2.

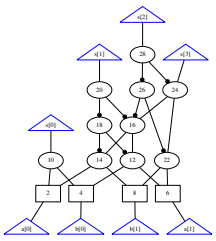
In contrast to more general polynomial division/reduction algorithms we use the fact in Alg. 2 that $\mathrm{lm}(p_v) = -v$. Because of the UMLT property and the fact that all leading coefficients of $G(C)$ are -1, Alg. 2 essentially boils down to substituting $v = \mathrm{lt}(p_v)$ by $\mathrm{tail}(p_v)$ in p in the case of circuit verification.

Algorithm 2 returns polynomials $h, r \in \mathbb{Z}[X]$ such that $p + hp_v = r \mod \langle B(X) \rangle \in \mathbb{Z}[X]$. We replace the polynomial p by the calculated remainder r [34]. To keep track of the rewriting steps we want to store information on the derivation of the rewritten polynomial r.

4.2 Generating PAC proofs

AMulet 2.0 generates PAC proofs as follows. The set of polynomials $G(C)$ determines the initial constraint set. The specification $\mathcal{L}$ defines the target polynomial of the proof. Proof steps have to be generated whenever polynomials are manipulated, that is during preprocessing for variable elimination and during reduction.

For variable elimination we produce proof steps which simulate reduction of a polynomial p by a polynomial p_v, cf. Alg. 2. Note that p and p_v are both con-



AIG node	Index	Polynomial equation
$l_{10} = b_0 \wedge a_0$	1	`-l10 + b0*a0;`
$l_{12} = b_0 \wedge a_1$	2	`-l12 + b0*a1;`
$l_{14} = b_1 \wedge a_0$	3	`-l14 + b1*a0;`
$l_{16} = l_{14} \wedge l_{12}$	4	`-l16 + l14*l12;`
$l_{18} = \neg l_{14} \wedge \neg l_{12}$	5	`-l18 + l14*l12-l14-l12+1;`
$l_{20} = \neg l_{18} \wedge \neg l_{16}$	6	`-l20 + l18*l16-l18-l16+1;`
$l_{22} = b_1 \wedge a_1$	7	`-l22 + b1*a1;`
$l_{24} = l_{22} \wedge l_{16}$	8	`-l24 + l22*l16;`
$l_{26} = \neg l_{22} \wedge \neg l_{16}$	9	`-l26 + l22*l16-l22-l16+1;`
$l_{28} = \neg l_{26} \wedge \neg l_{24}$	10	`-l28 + l26*l24-l26-l24+1;`
$s_0 = l_{10}$	11	`-s0 + l10;`
$s_1 = l_{20}$	12	`-s1 + l20;`
$s_2 = l_{28}$	13	`-s2 + l28;`
$s_3 = l_{24}$	14	`-s3 + l24;`

Fig. 3: AIG of a simple 2 bit multiplier in AIGER format (left) with induced constraint set (right).

tained in $G(C)$ and thus appear earlier in the proof. In general two proof steps are generated, a multiplication step and an addition step

$$id_i \texttt{ * } h\texttt{,}id_{p_v}\texttt{,}hp_v\texttt{;} \qquad id_{i+1} \texttt{ + } id_p\texttt{,}id_i\texttt{,}r\texttt{;}$$

where id_i and id_{i+1} define unused indices, and id_p and id_{p_v} represent the indices of polynomials p resp. p_v. The polynomial hp_v in above proof steps defines the expanded polynomial of multiplying $h \cdot p_v$ in $\mathbb{Z}[X]$. If $\mathrm{lt}(p_v) = v$ does not occur in any other polynomial $g \in G(C) \setminus \{p_v\}$, we can delete p_v from the constraint set, which we indicate by generating a deleting step

$$id_{p_v} \texttt{ d;}$$

After preprocessing is completed we gain the simplified polynomial model $G(C)'$. For monitoring the reduction of $\mathcal{L}$ by $G(C)'$ we have to generate proof steps which simulate the reduction of $\mathcal{L}$ by polynomials $g \in G(C)'$. We consider the polynomials $g \in G(C)'$ in the reverse topological order, such that each polynomial in $G(C)'$ has to be considered exactly once for reduction.

However in contrast to variable elimination, the specification $\mathcal{L}$, which acts as p in Alg. 2, is not part of the constraint set. Thus we are not able to simply generate two proof steps as before, because checking the addition rule would raise an error, as $p = \mathcal{L}$ does not occur earlier in the proof. On the other hand recall that all elements of an ideal can be represented as a linear combination of the generators of the ideal. To simulate the linear combination we generate a multiplication PAC step for each reduction step by a polynomial $g \in G(C)'$ and store the computed factor hg (h is the returned co-factor of Alg. 2). After reducing by several polynomials, we use a sequence of addition steps to gain a single intermediate specification polynomial. The reason for the intermediate summing up of polynomials is to keep the memory usage for proof generation small as we do not want to store too many factors at the same time. After reduction is completed we sum up all intermediate specifications. If the circuit is correct the final polynomial is the specification of the circuit.

```
15 * 5, l16-1, -l18*l16+l18+l16*l14*l12-l16*l14-l16*l12+l16-l14*l12+l14+l12-1;
16 + 6, 15, -l20+l16*l14*l12-l16*l14-l16*l12-l14*l12+l14+l12;
17 * 4, l14*l12-l14-l12, -l16*l14*l12+l16*l14+l16*l12-l14*l12;
18 + 16, 17, -l20-2*l14*l12+l14+l12;
19 * 9, l24-1, -l26*l24+l26+l24*l22*l16-l24*l22-l24*l16+l24-l22*l16+l22+l16-1;
20 + 10, 19, -l28+l24*l22*l16-l24*l22-l24*l16-l22*l16+l22+l16;
21 * 8, l22*l16-l22-l16, -l24*l22*l16+l24*l22+l24*l16-l22*l16;
22 + 20, 21, -l28-2*l22*l16+l22+l16;
23 * 14, 8, -8*s3+8*l24;
24 * 13, 4, -4*s2+4*l28;
25 * 22, 4, -4*l28-8*l22*l16+4*l22+4*l16;
26 + 25, 24, -4*s2-8*l22*l16+4*l22+4*l16;
27 * 8, 8, -8*l24+8*l22*l16;
28 * 7, 4, -4*l22+4*b1*a1;
29 + 28, 27, -8*l24+8*l22*l16-4*l22+4*b1*a1;
30 + 29, 26, -4*s2-8*l24+4*l16+4*b1*a1;
31 + 30, 23, -8*s3-4*s2+4*l16+4*b1*a1;
32 * 12, 2, -2*s1+2*l20;
33 * 18, 2, -2*l20-4*l14*l12+2*l14+2*l12;
34 + 32, 33, -2*s1-4*l14*l12+2*l14+2*l12;
35 * 4, 4, -4*l16+4*l14*l12;
36 * 3, 2, -2*l14+2*b1*a0;
37 + 35, 36, -4*l16+4*l14*l12-2*l14+2*b1*a0;
38 + 37, 34, -2*s1-4*l16+2*l12+2*b1*a0;
39 * 2, 2, -2*l12+2*b0*a1;
40 + 38, 39, -2*s1-4*l16+2*b1*a0+2*b0*a1;
41 + 1, 11, -s0+b0*a0;
42 + 40, 41, -2*s1-s0-4*l16+2*b1*a0+2*b0*a1+b0*a0;
43 + 42, 31, -8*s3-4*s2-2*s1-s0+4*b1*a1+2*b1*a0+2*b0*a1+b0*a0;
```

Fig. 4: Generating PAC steps during multiplier verification.

Example 10 Figure 3 shows an AIG of a simple 2-bit multiplier. For each node we introduce the corresponding polynomial equation. These polynomials are shown on the right side of Fig. 3 and define the initial constraint set. The multiplier is correct if we derive that the gate polynomials imply the specification $-8s_3 - 4s_2 - 2s_1 - s_0 + 4a_1b_1 + 2a_1b_0 + 2a_0b_1 + a_0b_0 = 0$.

The corresponding PAC proof can be seen in Fig. 4. Steps 15–22 are generated during preprocessing. The remaining steps are generated during reduction of the specification by $G(C)'$. The result of step 43 matches the circuit specification.

4.3 Generating NSS proofs

In this section we discuss how NSS proofs are generated in our verification tool AMULET 2.0. We introduced in the previous section that we distinguish two phases during verification of multipliers. In the preprocessing step we eliminate variables from $G(C)$ to gain a simpler polynomial representation $G(C)'$. In the second step the specification is reduced by $G(C)'$ to determine whether the given circuit is correct. Both phases have to be included in the NSS proof to yield a representation of the specification $\mathcal{L}$ as a linear combination of the original gate polynomials $G(C) \in \mathbb{Z}[X]$.

Definition 7 For a given set of polynomials $G \subset \mathbb{Z}[X]$, let $\text{base}(r) = \{(p_i, q_i) \mid p_i \in G,\ q_i \in \mathbb{Z}[X]\}$. We call $\text{base}(r)$ a basis representation of $r \in \mathbb{Z}[X]$ in terms of G, if there exist polynomials $v_1, \ldots v_l$ with $r = \sum_{(p_i,q_i)\ \in \text{base}(r)} q_i p_i + \sum_{i=1}^{l} v_i(x_i^2 - x_i)$.

Algorithm 3: Add-to-basis-representation($f, h,$ base(r))

Input : Polynomials f, $h \in \mathbb{Z}[X]$, basis representation base(r)
Output: Updated base(r) such that (f, h) is included

```
if base(f) = {(f, 1)} then
    if (f, h_i) ∈ base(r) for any h_i then
        base(r) ← (base(r) \ {(f, h_i)}) ∪ {(f, h_i + h)};
    else
        base(r) ← base(r) ∪ {(f, h)};
else
    foreach (f'_i, h'_i) ∈ base(f) do
        base(r) ← Add-to-basis-representation(f'_i, hh'_i, base(r))
return base(r)
```

To derive a NSS proof for $\mathcal{L}$ we aim to find a basis representation of $\mathcal{L}$ in terms of $G(C)$. For all polynomials $g \in G(C)$ it holds that base$(g) = \{(g, 1)\}$ is a basis representation in terms of $G(C)$.

As discussed in Sect. 4.1, we rewrite $G(C)$ by replacing polynomials of $G(C)$ by rewritten polynomials r that are derived using Alg. 2. To keep track of the rewriting steps we store information on the derivation of the rewritten polynomial r, i.e., we derive a basis representation of r in terms of $G(C)$. That is, we include the tuples $(p, 1)$, (p_v, h) as used in Alg. 2 in base(r).

Algorithm 3 shows how we update base(r) by adding a tuple (f, h). If the input polynomial f of Alg. 3 is an element of $G(C)$, i.e. base$(f) = \{(f, 1)\}$, we add the tuple (f, h) to base(r). If f does not occur in any tuple in base(r), we simply add (f, h) to base(r). Otherwise base(r) contains a tuple (f, h_i) that has to be updated to $(f, h_i + h)$, which corresponds to merging common factors in base(r).

If the polynomial f is not an original gate polynomial, f can be written as a linear combination $f = h'_1 f_1 + \cdots + h'_l f_l$ for some original polynomials f_i and $h'_i \in \mathbb{Z}[X]$. Thus the tuple (f, h) corresponds to $hf = hh'_1 f_1 + \cdots + hh'_l f_l$. We traverse through the tuples $(f_i, h'_i) \in$ base(f), multiply each of the co-factors h'_i by h and add the corresponding tuple (f_i, hh'_i) to base(r).

Multiplying and expanding the product hh_i may lead to an exponential blow-up in the size of the NSS proof as the following example shows.

Example 11 Consider OR-gates $y_0 = x_0 \vee x_1$, $y_1 = y_0 \vee x_2$, …, $y_k = y_{k-1} \vee x_{k+1}$ represented by the set of polynomials $G = \{-y_0 + x_0 + x_1 - x_0x_1, -y_1 + y_0 + x_2 - y_0x_2, \ldots, -y_k + y_{k-1} + x_{k+1} - y_{k-1}x_{k+1})\} \subseteq \mathbb{Z}[y_0, \ldots y_k, x_0, \ldots x_{k+1}]$. Assume we eliminate $y_1, \ldots, y_{k-1}$, yielding $y_k = x_0 \vee x_1 \vee \ldots \vee x_{k+1}$. The expanded polynomial representation of y_k contains 2^{k+2} monomials.

These sequences of OR-gates are common in carry-lookahead adders, which occur in complex multiplier architectures. This lead to the conjecture [31], which we stated in the introduction of this article. However, our previous verification approach [34] to tackle complex multipliers also relies on SAT solving. We substitute complex final-stage adders in multipliers by simple ripple-carry adders that do not rely on large OR-gates. Thus this blow-up does not occur in our experiments with our implementation (Sect. 6) for arithmetic circuit verification.

```
Constraint set                          Co-factors
-l10 + b0*a0                            1;
-l12 + b0*a1                            2;
-l14 + b1*a0                            2;
-l16 + l14*l12                          2*l14*l12-2*l14-2*l12+4;
-l18 + l14*l12 - l14 - l12 + 1          2*l16-2;
-l20 + l18*l16 - l18 - l16 + 1          2;
-l22 + b1*a1                            4;
-l24 + l22*l16                          4*l22*l16-4*l22-4*l16+8;
-l26 + l22*l16 - l22 - l16 + 1          4*l24-4;
-l28 + l26*l24 - l26 - l24 + 1          4;
 -s0 + l10                              1;
 -s1 + l20                              2;
 -s2 + l28                              4;
 -s3 + l24                              8;
```

Fig. 5: NSS proof for verifying the 2-bit multiplier that is depicted in Fig. 3.

Example 12 We demonstrate a sample run of Alg. 3. Let $G(C) = \{p_1, p_2, p_3\} \subseteq \mathbb{Z}[X]$ and $x, y, z \in \mathbb{Z}[X]$. Assume $q_1 = p_1 + xp_2$, $q_2 = p_3 + yp_2$, and their basis representations $\text{base}(q_1) = \{(p_1, 1), (p_2, x)\}$ and $\text{base}(q_2) = \{(p_2, y), (p_3, 1)\}$. Let $p = q_1 + zq_2$. We receive the basis representation of p in terms of $G(C)$ by adding $(q_1, 1)$ and (q_2, z) to $\text{base}(p)$.

$(q_1, 1)$: Since $q_1 \notin G(C)$, we add each tuple of $\text{base}(q_1) = \{(p_1, 1), (p_2, x)\}$ with co-factors multiplied by 1 to $\text{base}(p)$.

(q_2, z): We consider $\text{base}(q_2) = \{(p_2, y), (p_3, 1)\}$ and add (p_2, yz) and (p_3, z) to $\text{base}(p)$. Since p_3 is not yet contained in the ancestors of p, we directly add (p_3, z) to $\text{base}(p)$. The polynomial p_2 is already contained in $\text{base}(p)$, thus we add yz to the co-factor x of p_2 and we derive $\text{base}(p) = \{(p_1, 1), (p_2, x + yz), (p_3, z)\}$.

After preprocessing is completed, we repeatedly apply Alg. 2 and reduce the specification polynomial $\mathcal{L}$ by $G(C)'$. We generate the final NSS proof by deriving a basis representation for $\mathcal{L}$. Therefore we add after each reduction step the tuple (g, h), where h is the corresponding co-factor of polynomial g, to $\text{base}(\mathcal{L})$ using Alg. 3. After the final reduction step, $\text{base}(\mathcal{L})$ represents an NSS proof and is printed to a file.

Example 13 Figure 5 shows the corresponding NSS proof for the verification of the 2-bit multiplier that is depicted in Fig. 3. It can be seen that the proof contains only the (ordered) co-factors and thus is smaller than the extensive PAC proof.

4.4 Generating LPAC proofs

The LPAC format allows us to deliver dense PAC proofs. Thus, the proof generation is very similar as described in Sect. 4.2, with the difference being the level of compactness of the produced proof steps.

For each substitution step during preprocessing we generate a linear combination. That is, we merge the multiplication and addition steps, presented in Sect. 4.2 and gain for each preprocessing step a single step

$$id_i \texttt{ \% } id_{p_v}\texttt{*(}h\texttt{) + }id_p\texttt{, }r\texttt{;}$$

```
15 % 5*(l16-1) + 6, -l20+l16*l14*l12-l16*l14-l16*l12-l14*l12+l14+l12;
16 % 4*(l14*l12-l14-l12) + 15, -l20-2*l14*l12+l14+l12;
17 % 9*(l24-1) + 10, -l28+l24*l22*l16-l24*l22-l24*l16-l22*l16+l22+l16;
18 % 8*(l22*l16-l22-l16) + 17, -l28-2*l22*l16+l22+l16;
19 % 14*(8), -8*s3+8*l24;
20 % 7*(4) + 8*(8) + 18*(4) + 13*(4), -4*s2-8*l24+4*l16+4*b1*a1;
21 % 2*(2) + 3*(2) + 4*(4) + 16*(2) + 12*(2), -2*s1-4*l16+2*b1*a0+2*b0*a1;
22 % 1 + 11, -s0+b0*a0;
23 % 22 + 21 + 20 + 19, -8*s3-4*s2-2*s1-s0+4*b1*a1+2*b1*a0+2*b0*a1+b0*a0;
```

Fig. 6: Generating a LPAC proof during multiplier verification.

Similar as before, we generate deletion steps whenever p_v can be removed.

During the reduction phase we calculate and store the factors of each reduction step. After reducing by several polynomials we generate a linear combination step which sums up these factors to gain intermediate specifications. Thus, we are able to narrow down possible errors. Finally, we sum up the intermediate specifications in a single step and yield the specification $\mathcal{L}$.

Example 14 Figure 6 shows the corresponding LPAC proof for the verification of the 2-bit multiplier that is depicted in Fig. 3. The proof steps 15–18 are generated during preprocessing, 19–22 are generated during reduction and step with index 23 is the final step for summing up the intermediate specifications. It can be seen that LPAC enables merging PAC steps. For example the steps with indices 15 and 16 of Fig. 4 are now combined in the first proof step.

5 PAC Checkers

We have implemented two checkers for PAC proofs. The first, Pacheck, (Sect. 5.1) is efficient while the second, Pastèque, is verified using Isabelle/HOL (Sect. 5.2).

5.1 Pacheck 1.0

Pacheck consists of approximately 1 800 source lines of C code and is published [38] under MIT license. The default mode of Pacheck supports the extended version of PAC for the new syntax using indices. Pacheck also supports reasoning with exponents as described in the initial version of PAC. However, extension rules are only supported for Boolean models.

Pacheck reads three input files `<constraints>`, `<proof>`, and `<target>` and then verifies that the polynomial in `<target>` is contained in the ideal generated by the polynomials in `<constraints>` using the proof steps provided in `<proof>`. The polynomial arithmetic needed for checking the proof steps is implemented from scratch, because in the default setting we always calculate modulo the ideal $\langle B(X)\rangle$. General algorithms for polynomial arithmetic need to take exponent arithmetic over $\mathbb{Z}$ into account [55], which is not the case in our setting.

In the default mode of Pacheck we order variables in terms lexicographically using `strcmp`. All internally allocated terms are shared using a hash table. It turns out that the order of variables has an enormous effect on memory usage, since

Fig. 7: Term representation w.r.t. $v > u > x > y$ (left) and $x > u > y > v$ (right).

different variable orderings induce different terms. For example, given the monomials uxy and vxy. For the ordering $v > u > x > y$, the internal sharing is maximal and only 4 terms are allocated. For the ordering $x > u > y > v$, terms cannot be shared and thus 6 terms need to be allocated, cf. Fig. 7. For one example with more than 7 million proof steps, using `-1*strcmp` as sorting function leads to an increase of 50% in memory usage. A further option for sorting the variables is to use the variable appearance ordering from the given proof files. That is, we assign increasing `level` values to new variables during parsing of the proof file and sort according to this value. However, the best ordering that maximizes internal sharing cannot be determined in advance from the original constraint set, as it highly depends on the applied operations in the proof steps. PACHECK supports the orderings `strcmp`, `-1*strcmp`, `level`, and `-1*level`. Terms in polynomials are sorted using a lexicographic term order that is induced by the order of the variables.

Initially each polynomial from `<constraints>` is sorted and stored as an inference. Inferences consist of a given index and a polynomial and are stored in a hash table. Proof checking is applied on-the-fly. We parse each step of `<proof>` and immediately apply the necessary checks discussed in Sect. 2.1.3. If the proof step is either ADD or MULT, we have to compute whether the conclusion polynomial of the step is equal to the arithmetic operation performed on the antecedent polynomials.

Since the monomials of the polynomials are sorted, addition of polynomials is performed by merging their monomials in an interleaved way. Normalization of the exponents is not necessary in the ADD rule, but we still use this technique for multiplication, where we multiply each monomial of the first polynomial with each monomial of the second polynomial. In the MULT rule we normalize exponents larger than one, before testing equality. Furthermore, we check whether the conclusion polynomial of the ADD or MULT steps matches the polynomial in `<target>` to identify whether the normalized target polynomial was derived.

5.2 PASTÈQUE 1.0

To further increase trust in the verification, we implemented a verified checker called PASTÈQUE in the proof assistant Isabelle/HOL [52]. It follows a "refinement" approach, starting with an abstract specification of ideals, which we then refine with the Isabelle Refinement Framework [41] to the transition system from Sect. 2, and further down to executable code using Isabelle's code generator [18]. The Isabelle files have been made available [17]. The generated code consists of 2 800 lines Standard ML (2 400 generated by Isabelle, 400 for the parser) and is also available [17, 38] under MIT license.

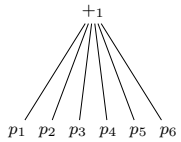
On the most abstract level, we start from Isabelle's definition of ideals. The specification states that if "success" is returned, the target is in the ideal. Then we formalize PAC and prove that the generated ideal is not changed by the proof steps. Proving that PAC respects the specification on ideals was not obvious due to limited automation and development of the Isabelle library of polynomials (e.g., "Var $(1) = \emptyset$" is not present). However, Sledgehammer [5] automatically proved many of these simple lemmas. We made a slightly different choice for definitions: Instead of using $B(X) = \{x^2 - x \mid x \in X\}$, we used $\{x^2 - x \mid \text{True}\}$ and proved that we only need variables of X. This made little difference for proofs, but avoided checking that variables are present in the problem.

While the input format identifies variables as strings, Isabelle only supports natural numbers as variables. Therefore, we use an injective function to convert between the abstract specification of polynomials (with natural numbers as variables) and the concrete manipulations (with strings as variables). The code does not depend on this function, only the correctness theorem does. Injectivity is only required to check that extension variables did not occur before.

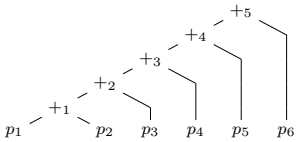
In the third refinement stage, Sepref [40] changes data structures automatically, such as replacing the set of variables X by a hash-set. Finally, we use the code generator to produce code. This code is combined with a trusted (unproven) parser and can be compiled using the Standard ML compiler MLton [59].

The implementation does not support the usage of exponents and is less sophisticated than Pacheck's. In particular, even if terms are sorted, sharing is not considered (neither of variables or of monomials) as it can be executed partially by the compiler, although not guaranteed by Standard ML semantics. Some sharing could also be performed by the garbage collector. We tried to enforce sharing by using MLton's `shareAll` function and by using a hash map during parsing, i.e., using a hash map that assigns a variable to "itself" (the same string, but potentially at a different memory location) and normalize every occurrence. However, performance became worse.

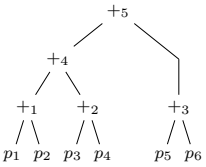
Pastèque is four times slower than Pacheck. First, this is due to Standard ML being intrinsically slower than C or C++. While Isabelle's code generator to LLVM [43] produces much faster code, we need integers of arbitrary large size, which is currently not supported. Also achieving sharing is entirely manual, which is challenging due to the use of separation logic Sepref. Second, there is no axiomatization of file reading and hence parsing must be applied *entirely* before calling the checker in order for the correctness theorem to apply. This is more memory intensive and less efficient than interleaving parsing and checking. Pastèque can be configured via the `uloop` option to either use the main loop generated by Isabelle (parsing before calling the generated checker) or instead use a hand-written copy of the main loop, the *unsafe loop*, where parsing and checking is interleaved. It is only unsafe because it is unchecked. However, the performance gain is large (on `sp-ar-cl-64` with 32 GB RAM, the garbage collection time went from 700 s down to 25 s), but only the checking functions of each step are verified, not the main loop.



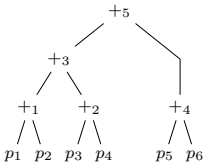
(a) Adding all polynomials at once



(b) Addition in sequence



(c) Tree structure, breadth first



(d) Tree structure, depth first

Fig. 8: Addition schemes of 6 polynomials.

6 The NSS Checker Nuss-Checker

Our NSS proof checker, Nuss-Checker is implemented in C. It consists of ~ 1500 source lines of code and is published [30] as open source under the MIT license. Similar to Pacheck, Nuss-Checker reads three input files `<constraints>`, `<cofact>`, and `<target>`. The file `<constraints>` contains the initial constraints $g_i \in G$, `<cofact>` contains the corresponding co-factors h_i in the same order. Nuss-Checker reads the files `<constraints>` and `<cofact>`, generates the products and then verifies that the sum of the products is equal to the polynomial f given in `<target>`. Nuss-Checker uses the same internal representation of polynomials as Pacheck and furthermore supports the same variables orders as Pacheck, with `strcmp` being the default ordering.

We validate the correctness of the generated NSS proofs by checking whether $\sum_{i=1}^{l} h_i g_i = f \in \mathbb{Z}[X]$ for $p_i \in G \subseteq \mathbb{Z}[X]$, $f, h_i \in \mathbb{Z}[X]$. This sounds rather straightforward as theoretically we only need to multiply the original constraints g_i by the co-factors h_i and calculate the sum of the products. However, we will discuss in this section that depending on the implementation the time and maximum amount of memory that is allocated varies by orders of magnitude.

Nuss-Checker generates the products $h_i g_i$ on the fly. That is, we parse both files `<constraints>` and `<cofact>` simultaneously, read two polynomials g_i and h_i from each file and calculate $h_i g_i$. Since addition of polynomials in $\mathbb{Z}[X]$ is associative, we are able to derive different addition schemes for n-ary addition. We experimented with five different addition/subtraction patterns. The addition patterns are depicted in Fig. 8 for adding six polynomials. The subscript i in "$+_i$" shows the order of the addition operation.

If we *sum up all polynomials at once*, we do not generate the intermediate addition results. Instead we push all monomials of the l products $h_i g_i$ onto one big

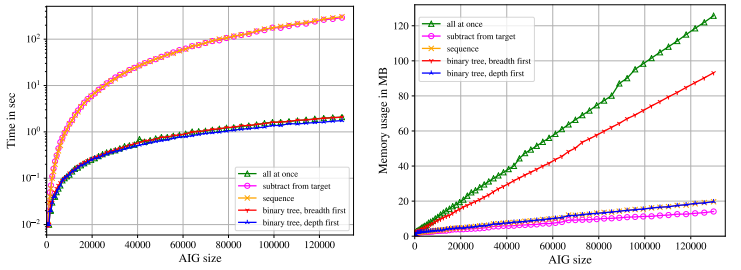


Fig. 9: Time (left) and memory usage (right) of addition schemes for btor multipliers.

stack. Afterwards, the monomials on the stack are sorted and merged, which corresponds to one big addition. However, all occurring monomials of the products are pushed on the stack and stored until the final sorting and merging, which increases the memory usage of Nuss-Checker.

If we *add up in sequence*, we only store one polynomial in the memory, and always add the latest product $h_i g_i$. On the one hand, this allows for monomials to cancel, which helps to reduce the memory usage. On the other hand, in the application of multiplier verification (cf. Sect. 4.1) the target polynomial $\mathcal{L}$ contains n^2 partial products $a_i b_j$ that lead to intermediate summands of quadratic size, which slows down the checking time.

For adding up in sequence we also experimented with the "inverse" operation, where we start with the target polynomial and step by step *subtract* the products $h_i g_i$ in the order originally used during the verification. We check whether the final polynomial is equal to zero. Again we always store only one polynomial in the memory, which admits a low memory usage. However, in our application the target polynomial is of quadratic size, making step-wise subtractions time-consuming.

If we add up in a *tree structure with breadth first*, we add two consecutive products of the NSS proof and store the resulting sum. After parsing the proof, we have $\frac{l}{2}$ polynomials on a stack. We repeatedly iterate over the stack and always sum up two consecutive polynomials, until only one polynomial is left. Using a tree addition scheme reduces the likelihood of quadratic sized intermediate summands for multiplier verification.

In the addition scheme, where we use a *tree structure and sum up depth first*, we develop the tree on-the-fly by always adding two polynomials of the same layer as soon as possible. It may be necessary to sum up remaining intermediate polynomials that are elements of different layers, as shown in Fig. 8. We always store at most $\lceil \log(l) \rceil$ polynomials in the memory, as a binary tree with l leafs has height $\lceil \log(l) \rceil$ and we never have more polynomials than layers in the memory.

We apply the presented addition schemes for our use case of multiplier verification. We choose two multiplier architectures. In our first experiment we consider a simple multiplier architecture, called *btor*, that is generated using Boolector [51] for various input sizes. Second, we examine a more complex multiplier architecture, called *bp-wt-rc*, that uses a Booth encoding and Wallace-tree accumulation. Fig-

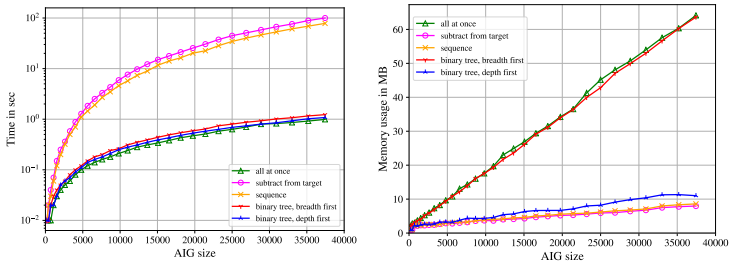


Fig. 10: Time (left) and memory usage (right) of addition schemes for bp-wt-rc multipliers.

ures 9 and 10 show that the results compare favorably to our conjectures of checking time and memory usage for each addition scheme. However, Nuss-Checker supports all presented options for addition, with *adding up in binary tree, depth first* set as default, because for different applications, using other addition schemes may be more beneficial. For example, we shuffled the order of the polynomials in the NSS proof of 128-bit btor-multipliers 200 times. The addition schemes "adding up in sequence" and "subtract" always exceeded the time limit of 300 seconds. The fastest addition scheme is "all at once", which is a factor of two faster than both tree-based addition schemes.

7 LPAC Checkers

The LPAC checkers combine the strength of PAC (checking intermediate steps and supporting extensions), while allowing doing a linear combination in a single step like NSS proofs. We have extended Pacheck (Sect. 7.1), based on our experiments for Nuss-Checker, and Pastèque (Sect. 7.2) to Pacheck 2.0 and Pastèque 2.0.

7.1 Pacheck 2

Pacheck 2.0 is a re-factorization and improved C++ reimplementation of our previous proof checkers. Since we are able to simulate PAC and NSS proofs in LPAC, Pacheck 2.0 unites and extends Pacheck 1.0 and Nuss-Checker.

The internal representation of polynomials is almost the same as for Pacheck 1.0. However, Pacheck 2.0 does no longer support the usage of exponents and thus only supports Boolean models. Proof checking is applied on the fly. That is, we parse a proof step and calculate that the linear combination of known polynomials is equal to the given conclusion polynomial of the proof step. We calculate linear combinations similar to proof checking a NSS proof in Nuss-Checker, i.e., whenever we parse a product of a polynomial and an index, we directly calculate the factor. The factors of the linear combination are processed using a *tree structure with depth first* addition scheme. Figure 11 shows a demonstration of Pacheck 2.0 on the LPAC proof of Ex. 14.

```
$ pacheck btor2.input btor2.proof btor2.target
[pck2] Pacheck Version 2.0
[pck2] Practical Algebraic Calculus Proof Checker
[pck2] Copyright(C) 2020, Daniela Kaufmann, Johannes Kepler University Linz
[pck2] sorting according to strcmp
[pck2] checking target enabled
[pck2] reading target polynomial from 'btor2.target'
[pck2] read 74 bytes from 'btor2.target'
[pck2]
[pck2] reading original polynomials from 'btor2.input'
[pck2] found 14 original polynomials in 'btor2.input'
[pck2] read 327 bytes from 'btor2.input'
[pck2]
[pck2] reading polynomial algebraic calculus proof from 'btor2.proof'
[pck2] found and checked 9 inferences in 'btor2.proof'
[pck2] read 680 bytes from 'btor2.proof'
[pck2]
[pck2] ------------------------------------------------------------------------
[pck2] c TARGET CHECKED
[pck2] ------------------------------------------------------------------------
[pck2]
[pck2] proof length:                    23 (total number of polynomials)
[pck2] proof size:                      82 (total number of monomials)
[pck2] proof degree:                     3
[pck2]
[pck2] total inferences:                23
[pck2] original inferences:             14 (61% of total rules)
[pck2] proof rules:                      9 (39% of total rules)
[pck2]   extensions:                     0 (0% of inference rules)
[pck2]   linear combination:             9 (100% of inference rules
[pck2]                                      containing 15 additions
[pck2]                                      and 14 multiplications)
[pck2] rules deleted:                    0 (0% of total rules)
[pck2]
[pck2] total allocated terms:           30
[pck2] max allocated terms:             30 (100% of total terms)
[pck2] searched terms:                 170 (82% hits,
[pck2]                                      0.0 average collisions)
[pck2] searched inferences:             69 (3.0 average searches,
[pck2]                                      0.0 average collisions)
[pck2]
[pck2] maximum resident set size:     2.67  MB
[pck2] process time:                  0.01  seconds
```

Fig. 11: Output of Pacheck 2.0 for the proof of Ex. 14.

7.2 Pastèque 2

Pastèque 2.0 [16] is developed on top of Pastèque 1.0. In order to reuse as much as possible from Pastèque 1.0, we reuse the specification and the rules of PAC. Instead of proving the correctness of the LPAC rules directly, we reduce them to the PAC rules, by seeing the LinComb rule as a series of Add and Mult. This requires the linear combination to not be empty: While 0 is always in the ideal, it cannot be generated by the PAC rules.

Additionally, we introduced explicit sharing of variables. We map every variable string to a unique 64-bit machine integer. In turn, this integer is the index of the original string in an array. Sharing is introduced in a new refinement step. The major change is that importing a new variable can now fail (if the problem contains more than 2^{64} different variables). This is nearly impossible in practical problems, but we had to add several new error paths in Pastèque. We obviously set up

the code generator to make the array access from machine words in an array without converting it to an unbounded integer. This change give us a performance improvement of around 10%, most likely because the memory representation is more efficient (fewer pointer indirections), making the work of the garbage collector easier.

On top of that, as we know that all our array accesses are valid (this is checked by SEPREF during synthesis of the code),[1] we add a flag such the compiler makes use of that. This also allowed us to use MLTON's LLVM backend that produce faster code, according to our experiments.

We did not change the implementation of the `uloop` option. Like PASTÈQUE 1.0, a full proof step is parsed before being checking. For NSS-style LPAC proof, this means that the full proof is still parsed before checking. In particular, for such proofs, PASTÈQUE 2.0 should be compared the default version of PASTÈQUE 1.0. The new sharing reduces memory usage, but parsing the full proof still causes a extreme memory pressure, as demonstrated by the experiments (Sect. 8). A solution would be to move the parsing to Isabelle (i.e., take a string as input instead of polynomials).

8 Experiments

In our experiments we use an Intel Xeon E5-2620 v4 CPU at 2.10 GHz (with turbo-mode disabled) with a memory limit of 128 GB. The time is listed in rounded seconds (wall-clock time). We measure the wall-clock time from starting the tools until they are finished. In our experiments we aim to provide a comprehensive comparison between our tools. Source code, benchmarks and experimental data are available [37].

8.1 PAC Proofs

For the experiments of Table 1 we generate PAC proofs as in previous work [34,35] to validate the correctness of multipliers with input bit-width n. The circuits are either generated with AMG [25], BOOLECTOR [51], or GENMUL [48].

For the upper part of Table 1 we generate proof certificates with our tool AMULET 2.0 [33] to validate the correctness of simple multiplier circuits. Our previous approach [34] to tackle complex multipliers also relies on SAT solving. We substitute complex final-stage adders in multipliers by simple ripple-carry adders. A bit-level miter is generated, which is passed on to a SAT solver to verify the equivalence of the adders. Computer algebra techniques are used to verify the rewritten multiplier. Since two different solving techniques are used, two proof certificates in distinct formats are generated. SAT solvers generate a DRUP proof and computer algebra techniques produce a PAC proof. In order to obtain a single proof certificate we translate DRUP proofs into PAC [35]. In the experiments of [35] all gate polynomials of the given multiplier, the equivalent ripple-carry adder, and the bit-level miter are assumed as initial set of constraints G. We even added polynomials that define Boolean negation to the initial constraint set. All these

[1] The hash map setup relies on exceptions, which is why we did not do that for PASTÈQUE 1.0, but now we changed the setup.

polynomials are now added using extension steps. This preserves the models of the gate polynomials of the given multiplier. Experiments for these proof certificates are shown in the lower part of Table 1. The second column shows the input bit-width and the third column shows the number of generated proof steps.

The memory usage for Pastèque depends on the garbage collector, which likely explains the peak around 64 GB, that is exactly half of the available memory, observed for the largest problems. Details on when and how the garbage collection trigger could explain the surprising bp-wt-cl where the `uloop` option uses more memory.

The effect of deletion rules and indices in Pacheck can also be seen in Table 1. In average deletion rules reduce the memory usage by ∼60%, with minimum 40% (for bp-ct-bk) and maximum 72% (for sp-ar-rc 512). Although the effect on runtime is limited. Using indices reduces the runtime by 30 to 80%. Note that in our earlier experiments [35] the proof checking time is slightly faster than in the column "no index", because we did not use proper extension rules, which requires the additional checks $p \in \mathbb{Z}[X]$ and $p^2 - p \equiv 0 \bmod \langle B(X) \rangle$.

8.2 LPAC and NSS

We have changed our pipeline to generate LPAC proofs instead of PAC proofs, using AMulet 2.0. The experiments are done on the same hardware. In the experiments of this section we only consider Pastèque with the `uloop` option.

We can only generate NSS proofs to validate the correctness of simple multiplier circuits that don't require combining algebra and SAT (i.e., extensions). It can be seen in Table 2 that NSS-style LPAC proofs are faster to check for Pacheck 2.0 than NSS proofs for Nuss-Checker. However, the memory usage of Pacheck 2.0 is around an order of magnitude higher than for Nuss-Checker, because Pacheck 2.0 reads and stores the complete constraint set before checking the proof. In Nuss-Checker the constraint set is parsed on the fly.

Pastèque 2.0 is very slow on NSS-style LPAC proofs because it must parse the entire file first, before starting checking, leading to very high memory usage. For those proofs, the `uloop` has no effect: A full proof step is parsed before checking, but since the entire proof is a single step, it is the same as parsing the full proof beforehand.

LPAC proofs (right block of Table 3) are checked as efficiently as NSS-style LPAC proofs (right block of Table 2) by Pacheck 2.0. For Pastèque 2.0 we gain a significant speed-up when using LPAC proofs. LPAC proofs only need between $1\% - 11\%$ of the corresponding checking time of NSS-style LPAC proofs. Additionally, checking LPAC proofs is more memory efficient.

If we compare checking LPAC proofs to checking PAC-style LPAC proofs, we can see that both Pacheck 2.0 and Pastèque 2.0 are a factor of two faster on checking LPAC proofs. The memory usage remains the same.

We further can see in Table 3 that both Pacheck 2.0 and Pastèque 2.0 are faster on LPAC proofs that simulate PAC than Pacheck 1.0 and Pastèque 1.0 on PAC proofs. The explicit sharing of variables in Pastèque 2.0 also significantly reduces the memory usage, except for sp-ar-rc 512 (the reasons for this behavior are unclear).

Table 1: Proof Checking (in bold the fastest version)

multiplier	n	steps	Pacheck 1.0						Pastèque 1.0			
			no delete		no index		default		default		`uloop`	
		(10^6)	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB
btor	128	0.3	**5**	273	11	100	**5**	92	22	3 886	17	1 773
btor	256	1.0	**25**	1 144	62	435	**25**	364	105	21 157	79	4 364
btor	512	4.2	**138**	4 956	402	1 972	141	1 461	531	64 412	416	22 292
sp-ar-rc	128	0.4	**6**	454	16	148	**6**	136	31	5 002	23	1 608
sp-ar-rc	256	1.6	29	1 858	96	651	**27**	541	139	32 525	102	8 769
sp-ar-rc	512	6.3	146	7 683	617	2 965	**134**	2 171	608	64 412	471	25 632
sp-ar-cl	32	1.6	23	773	36	354	**21**	353	121	40 654	113	9 492
sp-dt-lf	32	0.3	**2**	122	3	73	**2**	73	11	1 679	11	886
bp-ct-bk	32	0.2	**1**	86	2	52	**1**	51	8	1 600	7	1 068
bp-wt-cl	32	5.6	193	4 324	302	1 430	**181**	1 428	786	58 867	774	64 404

Table 2: NSS Proof Checking, without extension (in bold the fastest version)

multiplier	n	Nuss-Checker			LPAC simulates NSS				
		steps			steps	Pacheck 2.0		Pastèque 2.0	
			sec	MB		sec	MB	sec	MB
btor	128	1	2	18	1	**2**	98	53	2 044
btor	256	1	8	71	1	**7**	385	762	8 819
btor	512	1	41	295	1	**35**	1 555	14 347	41 712
sp-ar-rc	128	1	3	24	1	**2**	142	80	2 845
sp-ar-rc	256	1	13	95	1	**10**	561	1 181	12 275
sp-ar-rc	512	1	67	392	1	**48**	2 261	21 543	51 415

Table 3: LPAC Proof Checking (in bold the fastest version)

multiplier	n	LPAC simulates PAC					LPAC				
		steps	Pacheck 2.0		Pastèque 2.0		steps	Pacheck 2.0		Pastèque 2.0	
		(10^6)	sec	MB	sec	MB	(10^6)	sec	MB	sec	MB
btor	128	0.3	5	94	14	1 305	0.1	**2**	94	7	1 305
btor	256	1.3	26	367	67	3 467	0.3	**8**	367	37	3 816
btor	512	5.2	149	1 468	351	14 651	1.0	**37**	1 496	238	16 173
sp-ar-rc	128	0.4	5	137	15	1 330	0.1	**2**	137	8	1 330
sp-ar-rc	256	1.6	28	543	72	5 709	0.6	**11**	543	34	5 709
sp-ar-rc	512	6.3	145	2 174	381	34 327	2.4	**46**	2 173	180	34 327
sp-ar-cl	32	1.6	17	445	88	6 911	0.7	**10**	198	40	2 104
sp-dt-lf	32	0.3	2	80	8	857	0.2	**1**	39	4	383
bp-ct-bk	32	0.2	1	54	5	662	0.1	**1**	27	2	268
bp-wt-cl	32	5.5	144	2 250	646	36 224	2.4	**88**	1 094	292	10 489

Finally, we can compare the performance of Pacheck and Pastèque. In both versions, Pastèque 1.0 and Pastèque 2.0 is less efficient than Pacheck 1.0 and Pacheck 2.0. Pastèque is both much slower and more memory hungry. Verified checkers of SAT certificates [21, 42] have the same level of efficiency as state-of-the-art checkers [53], likely because of the imperative style (unlike our mostly functional code) and the more efficient memory usage by managing most memory directly (e.g., for clauses) instead of relying on the garbage collector.

9 Conclusion

In this article we presented the algebraic proof formats PAC, LPAC and NSS, which are able to validate algebraic verification results. We presented soundness and completeness arguments for these proof formats and showed how proof certificates can be generated as a by-product of algebraic reasoning on the use case of arithmetic circuit verification. Proofs in NSS capture whether a polynomial can be represented as a linear combination of a given set of polynomials by providing the co-factors of the linear combination. PAC proofs dynamically capture whether a polynomial can be derived providing a sequence of proof steps. We extend PAC by including an extension rule capturing rewriting techniques. Furthermore, we added a deletion rule and used indices for polynomials. Our novel format LPAC extends PAC by providing the ability to combine several steps at once.

Our proof checkers Pacheck, Pastèque, and Nuss-Checker are able to check proofs efficiently. Our experiments showed that the PAC optimizations cut the memory usage of Pacheck in half and reduce the runtime by around 30–80%. Our reimplementation Pacheck 2.0 and Pastèque 2.0, which use LPAC further reduce the runtime by around 25–50%. To our surprise, the size of NSS proofs does not explode in our experiments and is faster to check than PAC. This was the motivation to combine the advantages of PAC and NSS into LPAC. Checking LPAC proofs is as time efficient as checking NSS proofs, while still providing detailed error messages. However, the memory usage of checking LPAC proofs is an order of magnitude higher than checking pure NSS proofs. On LPAC, Pacheck was three times faster than Pastèque and used an order of magnitude less memory, whereas Pastèque was formally verified in Isabelle.

In the future we want to capture more general extension rules in PAC as the calculus from Section 2 allows. We imagine that it can be extended in two ways. First, we could relax the condition $p^2 = p$. This condition is necessary to have $v^2 = v$, but could be lifted even if it means that v^n cannot be simplified to v anymore, requiring to manipulate exponents. Second, we currently restrict the extension to the form $v = p$ where p contains no new variables. The correctness theorem does not rely on that and we leave it as future work to determine whether lifting one of these restrictions can lead to shorter proofs.

In AMulet 2.0 no redundant proof steps are generated, hence no backward proof checking is necessary unlike SAT certificates. This might still be interesting in other applications. Another idea for future work is to bridge the gap between C and Isabelle, either by imperative code or by verifying the C code directly.

References

1. Alekhnovich, M., Ben-Sasson, E., Razborov, A.A., Wigderson, A.: Space complexity in propositional calculus. SIAM J. Comput. **31**(4), 1184–1211 (2002)
2. Beame, P., Cook, S.A., Edmonds, J., Impagliazzo, R., Pitassi, T.: The relative complexity of NP search problems. J. Comput. Syst. Sci. **57**(1), 3–19 (1998)
3. Beame, P., Impagliazzo, R., Krajícek, J., Pitassi, T., Pudlák, P.: Lower Bounds on Hilbert's Nullstellensatz and Propositional Proofs. In: Proc. London Math. Society. vol. s3-73, pp. 1–26 (1996)
4. Becker, T., Weispfenning, V., Kredel, H.: Gröbner Bases. Springer (1993)
5. Blanchette, J.C., Böhme, S., Fleury, M., Smolka, S.J., Steckermeier, A.: Semi-intelligible Isar proofs from machine-generated proofs. J. Autom. Reasoning **56**(2), 155–200 (2016). doi:10.1007/s10817-015-9335-3
6. Bright, C., Kotsireas, I., Ganesh, V.: Applying computer algebra systems and SAT solvers to the Williamson conjecture. Journal of Symbolic Computation (04 2018). doi:10.1016/j.jsc.2019.07.024
7. Bright, C., Kotsireas, I., Ganesh, V.: SAT solvers and computer algebra systems: A powerful combination for mathematics. CoRR **abs/1907.04408** (2019)
8. Buchberger, B.: Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal. Ph.D. thesis, University of Innsbruck (1965)
9. Buss, S., Nordström, J.: Proof complexity and sat solving. Handbook of Satisfiability **336**, 233–350 (2021)
10. Choo, D., Soos, M., Chai, K.M.A., Meel, K.S.: Bosphorus: Bridging ANF and CNF solvers. In: Teich, J., Fummi, F. (eds.) DATE 2019. pp. 468–473. IEEE (2019). doi:10.23919/DATE.2019.8715061
11. Ciesielski, M.J., Su, T., Yasin, A., Yu, C.: Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model. IEEE TCAD pp. 1–1 (2019). doi:10.1109/TCAD.2019.2912944
12. Clegg, M., Edmonds, J., Impagliazzo, R.: Using the Groebner basis algorithm to find proofs of unsatisfiability. In: STOC. pp. 174–183. ACM (1996). doi:10.1145/237814.237860
13. Cox, D., Little, J., O'Shea, D.: Ideals, Varieties, and Algorithms. Springer-Verlag New York (1997)
14. Cruz-Filipe, L., Heule, M.J.H., Hunt, Jr., W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) CADE 26. LNCS, vol. 10395, pp. 220–236. Springer (2017). doi:10.1007/978-3-319-63046-5_14
15. Durán, A.J., Pérez, M., Varona, J.L.: The misfortunes of a trio of mathematicians using computer algebra systems. can we trust in them? Notices Amer. Math. Soc. **61**(10), 1249–1252 (2014)
16. Fleury, M., Kaufmann, D.: Isabelle PAC formalization, `http://people.mpi-inf.mpg.de/~mfleury/IsaFoL/current/PAC_Checker2/PAC_Checker/index.html`, theory files at `https://bitbucket.org/isafol/isafol/src/master/PAC_Checker2/`, Accessed: 2021-02-05
17. Fleury, M., Kaufmann, D.: Practical algebraic calculus checker. Arch. Formal Proofs **2020** (2020), `https://www.isa-afp.org/entries/PAC_Checker.html`
18. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer (2010). doi:10.1007/978-3-642-12251-4_9
19. Heule, M.J.H.: Computing small unit-distance graphs with chromatic number 5. CoRR **abs/1805.12181** (2018)
20. Heule, M.J.H., Biere, A.: Proofs for satisfiability problems. In: All about Proofs, Proofs for All. vol. 55, pp. 1–22 (2015). doi:10.1017/S1471068415000125
21. Heule, M.J.H., Hunt, Jr., W.A., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: ITP. LNCS, vol. 10499, pp. 269–284. Springer (2017). doi:10.1007/978-3-319-66107-0_18
22. Heule, M.J.H., Hunt, Jr., W.A., Wetzler, N.: Trimming while checking clausal proofs. In: FMCAD 2013. pp. 181–188. IEEE (2013), `http://ieeexplore.ieee.org/document/6679408/`
23. Heule, M.J.H., Kauers, M., Seidl, M.: Local search for fast matrix multiplication. In: SAT 2019. LNCS, vol. 11628, pp. 155–163. Springer (2019). doi:10.1007/978-3-030-24258-9_10
24. Heule, M.J.H., Kauers, M., Seidl, M.: New ways to multiply 3×3-matrices. Journal of Symbolic Computation **104**, 899–916 (2021)

25. Homma, N., Watanabe, Y., Aoki, T., Higuchi, T.: Formal design of arithmetic circuits based on arithmetic description language. IEICE Transactions **89-A**(12), 3500–3509 (2006)
26. Impagliazzo, R., Pudlák, P., Sgall, J.: Lower bounds for the polynomial calculus and the Gröbner basis algorithm. Computational Complexity **8**(2), 127–144 (1999)
27. Kapur, D.: Geometry theorem proving using Hilbert's Nullstellensatz. In: SYMSAC. pp. 202–208. ACM (1986). doi:10.1145/32439.32479
28. Kapur, D.: Using Gröbner bases to reason about geometry problems. J. Symb. Comput. **2**(4), 399–408 (1986). doi:10.1016/S0747-7171(86)80007-4
29. Kapur, D., Narendran, P.: An equational approach to theorem proving in first-order predicate calculus. In: IJCAI. pp. 1146–1153. Morgan Kaufmann (1985)
30. Kaufmann, D.: Nullstellensatz-proofs for multiplier verification, `http://fmv.jku.at/nussproofs`, accessed: 2021-02-05
31. Kaufmann, D.: Formal Verification of Multiplier Circuits using Computer Algebra. Ph.D. thesis, Informatik, Johannes Kepler University Linz (2020)
32. Kaufmann, D., Biere, A.: Nullstellensatz-proofs for multiplier verification. In: CASC. LNCS, vol. 12291, pp. 368–389. Springer (2020)
33. Kaufmann, D., Biere, A.: AMulet 2.0 for verifying multiplier circuits. In: TACAS (2). Lecture Notes in Computer Science, vol. 12652, pp. 357–364. Springer (2021)
34. Kaufmann, D., Biere, A., Kauers, M.: Verifying large multipliers by combining SAT and computer algebra. In: FMCAD 2019. pp. 28–36. IEEE (2019)
35. Kaufmann, D., Biere, A., Kauers, M.: From DRUP to PAC and back. In: DATE 2020. pp. 654–657. IEEE (2020), `http://fmv.jku.at/drup2pac/`
36. Kaufmann, D., Biere, A., Kauers, M.: SAT, Computer Algebra, Multipliers. In: Vampire 2018 and Vampire 2019. EPiC Series in Computing, vol. 71, pp. 1–18. EasyChair (2020)
37. Kaufmann, D., Fleury, M.: The LPAC checkers Pacheck 2.0 and Pastèque 2.0, `http://fmv.jku.at/lpac`, accessed: 2021-02-05
38. Kaufmann, D., Fleury, M.: The PAC checkers Pacheck and Pastèque, `http://fmv.jku.at/pacheck_pasteque`, accessed: 2021-02-05
39. Kaufmann, D., Fleury, M., Biere, A.: Pacheck and Pastèque, Checking Practical Algebraic Calculus Proofs. In: FMCAD 2020. FMCAD, vol. 1, pp. 264–269. TU Vienna Academic Press (2020), `http://fmv.jku.at/pacheck_pasteque/`
40. Lammich, P.: Refinement to Imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 253–269. Springer (2015). doi:10.1007/978-3-319-22102-1_17
41. Lammich, P.: Refinement based verification of imperative data structures. In: Avigad, J., Chlipala, A. (eds.) CPP 2016. pp. 27–36. ACM Press (2016). doi:10.1145/2854065.2854067
42. Lammich, P.: The GRAT tool chain - efficient (UN)SAT certificate checking with formal correctness guarantees. In: SAT. LNCS, vol. 10491, pp. 457–463. Springer (2017). doi:10.1007/978-3-319-66263-3_29
43. Lammich, P.: Generating verified LLVM from Isabelle/HOL. In: Tolmach, A., Harrison, J., O'Leary, J. (eds.) ITP 2019 (2019). doi:10.4230/LIPIcs.ITP.2019.22
44. Lv, J., Kalla, P., Enescu, F.: Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits. IEEE TCAD **32**(9), 1409–1420 (2013)
45. Mahzoon, A., Große, D., Drechsler, R.: PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers. In: Bahar, I. (ed.) ICCAD. p. 129. ACM (2018). doi:10.1145/3240765.3240837
46. Mahzoon, A., Große, D., Drechsler, R.: RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers. In: DAC. pp. 185:1–185:6. ACM (2019)
47. Mahzoon, A., Große, D., Scholl, C., Drechsler, R.: Towards formal verification of optimized and industrial multipliers. In: DATE 2020. pp. 544–549. IEEE (2020)
48. Mahzoon, A., Große, D., Drechsler, R.: Multiplier generator GenMul. `http://www.sca-verification.org/` (2019), accessed: 2021-02-05
49. Meir, O., Nordström, J., Robere, R., de Rezende, S.F.: Nullstellensatz size-degree tradeoffs from reversible pebbling. ECCC **137**, 18:1–18:16 (2019)
50. Miksa, M., Nordström, J.: A Generalized Method for Proving Polynomial Calculus Degree Lower Bounds. In: Conference on Computational Complexity, CCC 2015. LIPIcs, vol. 33, pp. 467–487. Schloss Dagstuhl (2015)
51. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , BtorMC and Boolector 3.0. In: CAV. LNCS, vol. 10981, pp. 587–595. Springer (2018)

52. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002). doi:10.1007/3-540-45949-9
53. Rebola-Pardo, A., Altmanninger, J.: Frying the egg, roasting the chicken: Unit deletions in DRAT proofs. In: Blanchette, J., Hritcu, C. (eds.) CPP. ACM (2020). doi:10.1145/3372885
54. Ritirc, D., Biere, A., Kauers, M.: A practical polynomial calculus for arithmetic circuit verification. In: Bigatti, A., Brain, M. (eds.) SC2'18. pp. 61–76. CEUR-WS (2018)
55. Roche, D.S.: What can (and can't) we do with sparse polynomials? In: ISSAC. pp. 25–30. ACM (2018)
56. Soos, M., Meel, K.S.: BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In: AAAI 2019. pp. 1592–1599. AAAI Press (2019). doi:10.1609/aaai.v33i01.33011592
57. Van Gelder, A.: Verifying RUP proofs of propositional unsatisfiability. In: ISAIM (2008)
58. Van Gelder, A.: Producing and verifying extremely large propositional refutations – have your cake and eat it too. Ann. Math. Artif. Intell. **65**(4), 329–372 (2012). doi:10.1007/s10472-012-9322-x
59. Weeks, S.: Whole-program compilation in MLton. In: Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006. p. 1. ACM Press (2006). doi:10.1145/1159876.1159877

10 Using SMT Proofs in Isabelle

Conference Paper

[1] Hans-Jörg Schurr, Fleury, Mathias, and Martin Desharnais. "Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant". In: *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*. Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. LNCS. I did the entire work Isabelle and most of the experiments and large parts of the writing. Springer, 2021, pp. 450–467.

Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant

Hans-Jörg Schurr[1], Mathias Fleury[2,3], and Martin Desharnais[4]

[1] University of Lorraine, CNRS, Inria, and LORIA, Nancy, France
hans-jorg.schurr@inria.fr
[2] Johannes Kepler University Linz, Linz, Austria
mathias.fleury@jku.at
[3] Max-Planck Institute für Informatik, Saarland Informatics Campus, Saarbrücken, Germany
[4] Universität der Bundeswehr München, München, Germany
martin.desharnais@unibw.de

Abstract. We present a fast and reliable reconstruction of proofs generated by the SMT solver veriT in Isabelle. The fine-grained proof format makes the reconstruction simple and efficient. For typical proof steps, such as arithmetic reasoning and skolemization, our reconstruction can avoid expensive search. By skipping proof steps that are irrelevant for Isabelle, the performance of proof checking is improved. Our method increases the success rate of Sledgehammer by halving the failure rate and reduces the checking time by 13%. We provide a detailed evaluation of the reconstruction time for each rule. The runtime is influenced by both simple rules that appear very often and common complex rules.

Keywords: automatic theorem provers · proof assistants · proof verification

1 Introduction

Proof assistants are used in verification and formal mathematics to provide trustworthy, machine-checkable formal proofs of theorems. Proof *automation* reduces the burden of finding proofs and allows proof assistant users to focus on the core of their arguments instead of technical details. A successful approach implemented by "hammers," like Sledgehammer for Isabelle [15], is to heuristically selects facts from the background; use an external automatic theorem prover, such as a satisfiability modulo theories (SMT) solver [12], to filter facts needed to discharge the goal; and to use the filtered facts to find a trusted proof.

Isabelle does not accept proofs that do not go through the assistant's inference kernel. Hence, Sledgehammer attempts to find the fastest internal method that can recreate the proof (*preplay*). This is often a call of the *smt* tactic, which runs an SMT solver, parses the proof, and reconstructs it through the kernel. This reconstruction allows the usage of external provers. The smt tactic was originally developed for the SMT solver Z3 [18, 34].

A. Platzer and G. Sutcliffe (Eds.): CADE 2021, LNAI 12699, pp. 450–467, 2021.
https://doi.org/10.1007/978-3-030-79876-5_26

The SMT solver CVC4 [10] is one of the best solvers on Sledgehammer generated problems [14], but currently does not produce proofs for problems with quantifiers. To reconstruct its proofs, Sledgehammer mostly uses the smt tactic based on Z3. However, since CVC4 uses more elaborate quantifier instantiation techniques, many problems provable for CVC4 are unprovable for Z3. Therefore, Sledgehammer regularly fails to find a trusted proof and the user has to write the proofs manually. veriT [19] (Sect. 2) supports these techniques and we extend the smt tactic to reconstruct its proofs. With the new reconstruction (Sect. 3), more smt calls are successful. Hence, less manual labor is required from users.

The runtime of the smt method depends on the runtime of the reconstruction and the solver. To simplify the reconstruction, we do not treat veriT as a black box anymore, but extend it to produce more detailed proofs that are easier to reconstruct. We use detailed rules for simplifications with a combination of propositional, arithmetic, and quantifier reasoning. Similarly, we add additional information to avoid search, e.g., for linear arithmetic and for term normalization. Our reconstruction method uses the newly provided information, but it also has a *step skipping* mode that combines some steps (Sect. 4).

A very early prototype of the extension was used to validate the fine-grained proof format itself [7, Sect. 6.2, second paragraph]. We also published some details of the reconstruction method and the rules [25] before adapting veriT to ease reconstruction. Here, we focus on the new features.

We optimize the performance further by tuning the search performed by veriT. Multiple options influence the execution time of an SMT solver. To fine-tune veriT's search procedure, we select four different combinations of options, or *strategies*, by generating typical problems and selecting options with complementary performance on these problems. We extend Sledgehammer to compare these four selected strategies and suggest the fastest to the user. We then evaluate the reconstruction with Sledgehammer on a large benchmark set. Our new tactic halves the failure rate. We also study the time required to reconstruct each rule. Many simple rules occur often, showing the importance of step skipping (Sect. 5).

Finally, we discuss related work (Sect. 6). Compared to the prototype [25], the smt tactic is now thoroughly tested. We fixed all issues revealed during development and improved the performance of the reconstruction method. The work presented here is integrated into Isabelle version 2021; i.e., since this version Sledgehammer can also suggest veriT, without user interaction. To simplify future reconstruction efforts, we document the proof format and all rules used by veriT. The resulting reference manual is part of the veriT documentation [40].

2 veriT and Proofs

The SMT solver veriT is an open source solver based on the CDCL($\mathcal{T}$) calculus. In proof-production mode, it supports the theories of uninterpreted functions with equality, linear real and integer arithmetic, and quantifiers. To support quantifiers veriT uses quantifier instantiation and extensive preprocessing.

veriT's proof syntax is an extension of SMT-LIB [11] which uses S-expressions and prefix notation. The proofs are refutation proofs, i.e., proofs of $\bot$. A proof is an indexed list of steps. Each step has a conclusion clause `(cl ..)` and is annotated with a rule, a list of premises, and some rule-dependent arguments. veriT distinguishes 90 rules [40]. Subproofs are the key feature of the proof format. They introduce an additional *context*. Contexts are used to reason about binders, e.g., preprocessing steps like transformation under quantifiers.

The conclusions of rules with contexts are always equalities. The context models a substitution into the free variables of the term on the left-hand side of the equality. Consider the following proof fragment that renames the variable name `x` to `vr`, as done during preprocessing:

```
(assume a0 (exists (x A) (f x))
(anchor :step t3 :args (:= x vr))
(step t1 (cl (= x vr)) :rule refl)
(step t2 (cl (= (f x) (f vr))) :rule cong :premises (t1))
(step t3 (cl (= (exists (x A) (f x))
                (exists (vr A) (f vr))) :rule bind)
```

The `assume` command repeats input assertions or states local assumptions. In this fragment the assumption `a0` is not used. Subproofs start with the `anchor` command that introduces a context. Semantically, the context is a shorthand for a lambda abstraction of the free variable and an application of the substituted term. Here the context is $x \mapsto \text{vr}$ and the step `t1` means $(\lambda x.\ x)\ \text{vr} = \text{vr}$. The step is proven by congruence (rule `cong`). Then congruence is applied again (step `t2`) to prove that $(\lambda x.\ f\ x)\ \text{vr} = f\ \text{vr}$ and step `t3` concludes the renaming.

During proof search each module of veriT appends steps onto a list. Once the proof is completed, veriT performs some cleanup before printing the proof. First, a pruning phase removes branches of the proof not connected to the root $\bot$. Second, a merge phase removes duplicated steps. The final pass prepares the data structures for the optional term sharing via name annotations.

3 Overview of the veriT-Powered smt Tactic

Isabelle is a generic proof assistant based on an intuitionistic logic framework, *Pure*, and is almost always only used parameterized with a logic. In this work we use only Isabelle/HOL, the parameterization of Isabelle with higher-order logic with rank-1 (top level) polymorphism. Isabelle adheres to the LCF [26] tradition. Its kernel supports only a small number of inferences. Tactics are programs that prove a goal by using only the kernel for inferences. The LCF tradition also means that external tools, like SMT solvers, are not trusted.

Nevertheless, external tools are successfully used. They provide relevant facts or a detailed proof. The Sledgehammer tool implements the former and passes the filtered facts to trusted tactics during preplay. The smt tactic implements the latter approach. The provided proof is checked by Isabelle. We focus on the smt tactic, but we also extended Sledgehammer to also suggest our new tactic.

The smt tactic translates the current goal to the SMT-LIB format [11], runs an SMT solver, parses the proof, and replays it through Isabelle's kernel. To choose the smt tactic the user applies `(smt (z3))` to use Z3 and `(smt (verit))` to use veriT. We will refer to them as z-smt and v-smt. The proof formats of Z3 and veriT are so different that separate reconstruction modules are needed. The v-smt tactic performs four steps:

1. It negates the proof goal to have a refutation proof and also encodes the goal into first-order logic. The encoding eliminates lambda functions. To do so, it replaces each lambda function with a new function and creates app operators corresponding to function application. Then veriT is called to find a proof.
2. It parses the proof found by veriT (if one is found) and encodes it as a directed acyclic graph with $\bot$ as the only conclusion.
3. It converts the SMT-LIB terms to typed Isabelle terms and also reverses the encoding used to convert higher-order into first-order terms.
4. It traverses the proof graph, checks that all input assertions match their Isabelle counterpart and then reconstructs the proof step by step using the kernel's primitives.

4 Tuning the Reconstruction

To improve the speed of the reconstruction method, we create small and well-defined rules for preprocessing simplifications (Sect. 4.1). Previously, veriT implicitly normalized every step; e.g., repeated literals were immediately deleted. It now produces proofs for this transformation (Sect. 4.2). Finally, the linear-arithmetic steps contain coefficients which allow Isabelle to reconstruct the step without relying on its limited arithmetic automation (Sect. 4.3). On the Isabelle side, the reconstruction module selectively decodes the first-order encoding (Sect. 4.4). To improve the performance of the reconstruction, it skips some steps (Sect. 4.5).

4.1 Preprocessing Rules

During preprocessing SMT solvers perform simplifications on the operator level which are often akin to simple calculations; e.g., $a \times 0 \times f(x)$ is replaced by 0.

To capture such simplifications, we create a list of 17 new rules: one rule per arithmetic operator, one to replace boolean operators such as XOR with their definition, and one to replace n-ary operator applications with binary applications. This is a compromise: having one rule for every possible simplification would create a longer proof. Since preprocessing uses structural recursion, the implementation simply picks the right rule in each leaf case. The example above now produces a `prod_simplify` step with the conclusion $a \times 0 \times f(x) = 0$. Previously, a single step of the `connect_equiv` rule collected all those simplifications and no list of simplifications performed by this rule existed. The reconstruction relied an experimentally created list of tactics to be fast enough.

On the Isabelle side, the reconstruction is fast, because we can direct the search instead of trying automated tactics that can also work on other parts of

the formula. For example, the simplifier handles the numeral manipulations of the `prod_simplify` rule and we restrict it to only use arithmetic lemmas.

Moreover, since we know the performed transformations, we can ignore some parts of the terms by *generalizing*, i.e., replacing them by constants [18]. Because generalized terms are smaller, the search is more directed and we are less likely to hit the search-depth limitation of Isabelle's `auto` tactic as before. Overall, the reconstruction is more robust and easier to debug.

4.2 Implicit Steps

To simplify reconstruction, we avoid any implicit normal form of conclusions. For example, a rule concluding $t \vee P$ for any formula t can be used to prove $P \vee P$. In such cases veriT automatically normalizes the conclusion $P \vee P$ to P. Without a proof of the normalization, the reconstruction has to handle such cases.

We add new proof rules for the normalization and extend veriT to use them. Instead of keeping only the normalized step, both the original and the normalized step appear in the proof. For the example above, we have the step $P \vee P$ and the normalized P. To remove a double negation $\neg\neg t$ we introduce the tautology $\neg\neg\neg t \vee t$ and resolve it with the original clause. Our changes do not affect any other part of veriT. The solver now also prunes steps concluding $\top$.

On the Isabelle side, the reconstruction becomes more regular with fewer special cases and is more reliable. The reconstruction method can directly reconstruct rules. To deal with the normalization, the reconstruction used to first generate the conclusion of the theorem and then ran the simplifier to match the normalized conclusion. This could not deal with tautologies.

We also improve the proof reconstruction of quantifier instantiation steps. One of the instantiation schemes, *conflicting instances* [8,36], only works on clausified terms. We introduce an explicit quantified-clausification rule `qnt_cnf` issued before instantiating. While this rule is not detailed, knowing when clausification is needed improves reconstruction, because it avoids clausifying unconditionally. The clausification is also shared between instantiations of the same term.

4.3 Arithmetic Reasoning

We use a proof witness to handle linear arithmetic. When the propositional model is unsatisfiable in the theory of linear real arithmetic, the solver creates `la_generic` steps. The conclusion is a tautological clause of linear inequalities and equations and the justification of the step is a list of coefficients so that the linear combination is a trivially contradictory inequality after simplification (e.g., $0 \geqslant 1$). Farkas' lemma guarantees the existence of such coefficients for reals. Most SMT solvers, including veriT, use the simplex method [21] to handle linear arithmetic. It calculates the coefficients during normal operation.

The real arithmetic solver also strengthens inequalities on integer variables before adding them to the simplex method. For example, if x is an integer the inequality $2x < 3$ becomes $x \leqslant 1$. The corresponding justification is the rational coefficient $1/2$. The reconstruction must replay this strengthening.

The complete linear arithmetic proof step $1 < x \vee 2x < 3$ looks like

```
(step t11 (cl (< 1 x) (< (* 2 x) 3))
    :rule la_generic :args (1 (div 1 2)))
```

The reconstruction of an `la_generic` step in Isabelle starts with the goal $\bigvee_i \neg c_i$ where each c_i is either an equality or an inequality. The reconstruction method first generalizes over the non-arithmetic parts. Then it transforms the lemma into the equivalent formulation $c_1 \Rightarrow \cdots \Rightarrow c_n \Rightarrow \bot$ and removes all negations (e.g., by replacing $\neg a \leqslant b$ with $b > a$).

Next, the reconstruction method multiplies the equation by the corresponding coefficient. For example, for integers, the equation $A < B$, and the coefficient p/q (with $p > 0$ and $q > 0$), it strengthens the equation and multiplies by p to get

$$p \times (A \mathit{div}\, q) + p \times (\text{if } B \mathit{mod}\, q = 0 \text{ then } 1 \text{ else } 0) \leqslant p \times (B \mathit{div}\, q).$$

The if-then-else term (if $B \mathit{mod}\, q = 0$ then 1 else 0) corresponds to the strengthening. If $B \mathit{mod}\, q = 0$, the result is an equation of the form $A' + 1 \leqslant B'$, i.e., $A' < B'$. No strengthening is required for the corresponding theorem over reals.

Finally, we can combine all the equations by summing them while being careful with the equalities that can appear. We simplify the resulting (in)equality using Isabelle's simplifier to derive $\bot$.

To replay linear arithmetic steps, Isabelle can also use the tactic linarith as used for Z3 proofs. It searches the coefficients necessary to verify the lemma. The reconstruction used it previously [25], but the tactic can only find integer coefficients and fails if strengthening is required. Now the rule is a mechanically checkable certificate.

4.4 Selective Decoding of the First-order Encoding

Next, we consider an example of a rule that shows the interplay of the higher-order encoding and the reconstruction. To express function application, the encoding introduces the first-order function app and constants for encoded functions. The proof rule `eq_congruent` expresses congruence on a first-order function: $(t_1 \neq u_1) \vee \ldots \vee (t_n \neq u_n) \vee f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n)$. With the encoding it can conclude $f \neq f' \vee x \neq x' \vee \mathrm{app}(f, x) = \mathrm{app}(f', x')$. If the reconstruction unfolds the entire encoding, it builds the term $f \neq f' \vee x \neq x' \vee fx = f'x'$. It then identifies the functions and the function arguments and uses rewriting to prove that if $f = f'$ and $x = x'$, then $fx = f'x'$.

However, Isabelle β-reduces all terms implicitly, changing the term structure. Assume $f := \lambda x.\ x = a$ and $f' := \lambda x.\ a = x$. After unfolding all constructs that encode higher-order terms and after β-reduction, we get $(\lambda x.\ x = a) \neq (\lambda x.\ a = x') \vee (x \neq x') \vee (x = a) = (a = y')$. The reconstruction method cannot identify the functions and function arguments anymore.

Instead, the reconstruction method does not unfold the encoding including app. This eliminates the need for a special case to detect lambda functions. Such a case was used in the previous prototype, but the code was very involved and hard to test (such steps are rarely used).

4.5 Skipping Steps

The increased number of steps in the fine-grained proof format slows down reconstruction. For example, consider skolemization from $\exists x.\ P\ x$. The proof from Z3 uses *one* step. veriT uses *eight* steps—first renaming it to $(\exists x.\ P\ x) = (\exists v.\ P\ v)$ (with a subproof of at least 2 steps), then concluding the renaming to get $(\exists v.\ P\ v)$ (two steps), then $(\exists v.\ P\ v) = P\ (\epsilon v.\ P\ v)$ (with a subproof of at least 2 steps), and finally $P\ (\epsilon v.\ P\ v)$ (two steps).

To reduce the number of steps, our reconstruction skips two kinds of steps. First, it replaces every usage of the `or` rule by its only premise. Second, it skips the renaming of bound variables. The proof format treats $\forall x.\ P\ x$ and $\forall y.\ P\ y$ as two different terms and requires a detailed proof of the conversion. Isabelle, however, uses De Bruijn indices and variable names are irrelevant. Hence, we replace steps of the form $(\forall x.\ P\ x) \Leftrightarrow (\forall y.\ P\ y)$ by a single application of reflexivity. Since veriT canonizes all variable names, this eliminates many steps.

We can also simplify the idiom "`equiv_pos2`; `th_resolution`". veriT generates it for each skolemization and variable renaming. Step skipping replaces it by a single step which we replay using a specialized theorem.

On proof with quantifiers, step skipping can remove more than half of the steps—only four steps remain in the skolemization example above (where two are simply reflexivity). However, with step skipping the smt method is not an independent checker that confirms the validity of every single step in a proof.

5 Evaluation

During development we routinely tested our proof reconstruction to find bugs. As a side effect, we produced SMT-LIB files corresponding to the calls. We measure the performance of veriT with various options on them and select five different strategies (Sect. 5.1). We also evaluate the repartition of the tactics used by Sledgehammer for preplay (Sect. 5.2), and the impact of the rules (Sect. 5.3).

We performed the strategy selection on a computer with two Intel Xeon Gold 6130 CPUs (32 cores, 64 threads) and 192 GiB of RAM. We performed Isabelle experiments with Isabelle version 2021 on a computer with two AMD EPYC 7702 CPUs (128 cores, 256 threads) and 2 TiB of RAM.

5.1 Strategies

veriT exposes a wide range of options to fine-tune the proof search. In order to find good combinations of options (*strategies*), we generate problems with Sledgehammer and use them to fine-tune veriT's search behavior. Generating problems also makes it possible to test and debug our reconstruction.

We test the reconstruction by using Isabelle's *Mirabelle* tool. It reads theories and automatically runs Sledgehammer [14] on all proof steps. Sledgehammer calls various automatic provers (here the SMT solvers CVC4, veriT, and Z3 and the superposition prover E [38]) to *filter* facts and chooses the fastest tactic that can prove the goal. The tactic smt is used as a last resort.

Table 1. Options corresponding to the different veriT strategies

Name	Options
default	(no option)
del_insts	--index-sorts --index-fresh-sorts --ccfv-breadth --inst-deletion --index-SAT-triggers --inst-deletion-loops --inst-deletion-track-var
ccfv_SIG	--triggers-new --index-SIG --triggers-sel-rm-specific
ccfv_insts	--triggers-new --index-sorts --index-fresh-sorts --triggers-sel-rm-specific --triggers-restrict-combine --inst-deletion-loops --index-SAT-triggers --inst-deletion-track-vars --ccfv-index=100000 --ccfv-index-full=1000 --inst-sorts-threshold=100000 --ematch-exp=10000000 --inst-deletion
best	--triggers-new --index-sorts --index-fresh-sorts --triggers-sel-rm-specific

To generate problems for tuning veriT, we use the theories from HOL-Library (an extended standard library containing various developments) and from the formalizations of Green's theorem [2,3], the Prime Number Theorem [23], and the KBO ordering [13]. We call Mirabelle with only veriT as a fact filter. This produces SMT files for representative problems Isabelle users want to solve and a series of calls to v-smt. For failing v-smt calls three cases are possible: veriT does not find a proof, reconstruction times out, or reconstruction fails with an error. We solved all reconstruction failures in the test theories.

To find good strategies, we determine which problems are solved by several combination of options within a two second timeout. We then choose the strategy which solves the most benchmarks and three strategies which together solve the most benchmarks. For comparison, we also keep the default strategy.

The strategies are shown in Table 1 and mostly differ in the instantiation schemes. The strategy *del_insts* uses instance deletion [6] and uses a breadth-first algorithm to find conflicting instances. All other strategies rely on extended trigger inference [29]. The strategy *ccfv_SIG* uses a different indexing method for instantiation. It also restricts enumerative instantiation [35], because the options --index-sorts and --index-fresh-sorts are not used. The strategy *ccfv_insts* increases some thresholds. Finally, the strategy *best* uses a subset of the options used by the other strategies. Sledgehammer uses *best* for fact filtering.

We have also considered using a scheduler in Isabelle as used in the SMT competition. The advantage is that we do not need to select the strategy on the Isabelle side. However, it would make v-smt unreliable. A problem solved by only one strategy just before the end of its time slice can become unprovable on slower hardware. Issues with z-smt timeouts have been reported on the Isabelle mailing list, e.g., due to an antivirus delaying the startup [27].

5.2 Improvements of Sledgehammer Results

To measure the performance of the v-smt tactic, we ran Mirabelle on the full HOL-Library, the theory Prime Distribution Elementary (PDE) [22], an executable resolution prover (RP) [37], and the Simplex algorithm [30]. We extended Sledgehammer's proof preplay to try all veriT strategies and added instrumentation for

Table 2. Outcome of Sledgehammer calls showing the total success rate (SR, higher is better) of one-liner proof preplay, the number of suggested v-smt (OL_v) and z-smt (OL_z) one-liners, and the number of preplay failures (PF, lower is better), in percentages of the unique goals.

	HOL-Library (13 562 goals)				PNT (1 715 goals)				RP (1 658 goals)				Simplex (1 982 goals)			
	SR	OL_v	OL_z	PF	SR	OL_v	OL_z	PF	SR	OL_v	OL_z	PF	SR	OL_v	OL_z	PF
Fact-filter prover: CVC4																
z-smt	54.5		2.7	1.5	33.1		3.7	0.8	64.8		1.3	0.8	51.6		1.6	0.9
both	55.5	2.5	1.1	0.5	33.6	3.6	0.6	0.3	65.3	1.4	0.4	0.3	52.1	1.1	1.0	0.4
Fact-filter prover: E																
z-smt	55.5		1.1	1.7	36.0		0.3	1.7	61.7		0.7	1.2	49.8		1.4	0.7
both	56.0	0.8	0.7	1.3	36.4	0.6	0.1	1.3	62.1	0.9	0.2	0.8	49.9	0.3	1.3	0.5
Fact-filter prover: veriT																
z-smt	48.5		1.7	1.2	26.1		1.5	0.5	58.2		0.9	0.7	46.7		0.9	1.0
both	49.4	1.6	0.9	0.4	26.5	1.4	0.4	0.2	58.6	1.1	0.3	0.2	47.4	1.0	0.6	0.3
Fact-filter prover: Z3																
z-smt	50.8		2.5	0.8	27.9		2.7	0.4	60.4		0.8	0.7	48.3		0.9	0.3
both	51.3	1.9	1.1	0.3	28.2	2.5	0.5	0.1	60.9	1.1	0.1	0.2	48.4	0.4	0.6	0.2

the time of all tried tactics. Sledgehammer and automatic provers are mostly nondeterministic programs. To reduce the variance between the different Mirabelle runs, we use the deterministic MePo fact filter [33] instead of the better performing MaSh [28] that uses machine learning (and depends on previous runs) and underuse the hardware to minimize contention. We use the default timeouts of 30 seconds for the fact filtering and one second for the proof preplay. This is similar to the Judgment Day experiments [17]. The raw results are available [1].

Success Rate. Users are not interested in which tactics are used to prove a goal, but in how often Sledgehammer succeeds. There are three possible outcomes: (i) a successfully preplayed proof, (ii) a proof hint that failed to be preplayed (usually because of a timeout), or (iii) no proof. We define the success rate as the proportion of outcome (i) over the total number of Sledgehammer calls.

Table 2 gathers the results of running Sledgehammer on all unique goals and analyzing its outcome using different preplay configurations where only z-smt (the baseline) or both v-smt and z-smt are enabled. Any useful preplay tactic should increase the success rate (SR) by preplaying new proof hints provided by the fact-filter prover, reducing the preplay failure rate (PF).

Let us consider, e.g., the results when using CVC4 as fact-filter prover. The success rate of the baseline on the HOL-Library is 54.5% and its preplay failure rate is 1.5%. This means that CVC4 found a proof for $54.5\% + 1.5\% = 56\%$ of the goals, but that Isabelle's proof methods failed to preplay many of them. In such

cases, Sledgehammer gives a proof hint to the user, which has to manually find a functioning proof. By enabling v-smt, the failure rate decreases by two thirds, from 1.5% to 0.5%, which directly increases the success rate by 1 percentage point: new cases where the burden of the proof is moved from the user to the proof assistant. The failure rate is reduced in similar proportions for PNT (63%), RP (63%), and Simplex (56%). For these formalizations, this improvement translates to a smaller increase of the success rate, because the baseline failure rate was smaller to begin with. This confirms that the instantiation technique *conflicting instances* [8, 36] is important for CVC4.

When using veriT or Z3 as fact-filter prover, a failure rate of zero could be expected, since the same SMT solvers are used for both fact filtering and preplaying. The observed failure rate can partly be explained by the much smaller timeout for preplay (1 second) than for fact filtering (30 seconds).

Overall, these results show that our proof reconstruction enables Sledgehammer to successfully preplay more proofs. With v-smt enabled, the weighted average failure rate decreases as follows: for CVC4, from 1.3% to 0.4%; for E, from 1.5% to 1.2%; for veriT, from 1.0% to 0.3%; and for Z3, from 0.7% to 0.3%. For the user, this means that the availability of v-smt as a proof preplay tactic increases the number of goals that can be fully automatically proved.

Saved time. Table 3 shows a different view on the same results. Instead of the raw success rate, it shows the time that is spent reconstructing proofs. Using the baseline configuration, preplaying all formalizations takes a total of 250.1 + 33.4 + 37.2 + 42.8 = 363.5 seconds. When enabling v-smt, some calls to z-smt are replaced by faster v-smt calls and the reconstruction time decreases by 13% to 212.6 + 28.4 + 34.4 + 41.6 = 317 seconds. Note that the per-formalization improvement varies considerably: 15% for HOL-Library, 15% for PNT, 7.5% for RP, and 4.0% for Simplex.

For the user, this means that enabling v-smt as a proof preplay tactic may significantly reduce the verification time of their formalizations.

Impact of the Strategies. We have also studied what happens if we remove a single veriT strategy from Sledgehammer (Table 4). The most important one is *best*, as it solves the highest number of problems. On the contrary, *default* is nearly entirely covered by the other strategies. *ccfv_SIG* and *del_insts* have a similar number where they are faster than Z3, but the latter has more unique goals and therefore, saves more time. Each strategy has some uniquely solved problems that cannot be reconstructed using any other. The results are similar for the other theories used in Table 3.

5.3 Speed of Reconstruction

To better understand what the key rules of our reconstruction are, we recorded the time used to reconstruct each rule and the time required by the solver over all calls attempted by Sledgehammer including the ones not selected. The reconstruction ratio (reconstruction over search time) shows how much slower reconstructing

Table 3. Preplayed proofs (Pr.) and their execution time (s) when using CVC4 as fact-filter prover. Shared proofs are found with and without v-smt and new proofs are found only with v-smt. The proofs and their associated timings are categorized in one-liners using v-smt (OL_v), z-smt (OL_z), or any other Isabelle proof methods (OL_o).

		Total	Shared proofs				New proofs
			Total =	OL_v +	OL_z +	OL_o	OL_v
		Pr.	Time =	Time (Pr.) +	Time (Pr.) +	Time (Pr.)	Time (Pr.)
HOL-Library	z-smt	7 409	250.1 =		85.0 (362) +	165.1 (7 047)	
	both	7 545	212.6 =	27.9 (211) +	19.6 (152) +	165.1 (7 047)	34.7 (135)
PNT	z-smt	569	33.4 =		14.8 (64) +	18.5 (505)	
	both	577	28.4 =	7.7 (54) +	2.1 (10) +	18.5 (505)	3.4 (8)
RP	z-smt	1 077	37.2 =		8.7 (22) +	28.5 (1 055)	
	both	1 085	34.4 =	4.5 (16) +	1.4 (6) +	28.5 (1 055)	2.2 (8)
Simplex	z-smt	1 024	42.8 =		6.7 (32) +	36.0 (992)	
	both	1 033	41.6 =	2.4 (13) +	3.2 (19) +	36.0 (992)	3.0 (9)

Table 4. Reconstruction time and number of solved goals when removing a single strategy (HOL-Library results only), using CVC4 as fact filter.

	Shared proofs				New proofs	
	OL_v		OL_z		OL_v	
	Time	Proofs	Time	Proofs	Time	Proofs
No *best*	16.5	119	50.6	244	25.9	94
No *ccfv_SIG*	27.0	198	22.6	164	33.5	123
No *ccfv_threshold*	28.3	211	19.6	152	33.9	130
No *del_insts*	27.4	201	21.8	162	32.9	124
No *default*	27.9	207	20.1	156	33.8	134
Baseline	27.9	211	19.6	152	34.7	135

compared to finding a proof is. For the 25% of the proofs, Z3's concise format is better and the reconstruction is faster than proof finding (first quartile: 0.9 for v-smt vs. 0.1 for z-smt). The 99th percentile of the proofs (18.6 vs. 27.2) shows that veriT's detailed proof format reduces the number of slow proofs. The reconstruction is slower than finding proofs on average for both solvers.

Fig. 1 shows the distribution of the time spent on some rules. We remove the slowest and fastest 5% of the applications, because garbage collection can trigger at any moment and even trivial rules can be slow. Fig. 2 gives the sum of all reconstruction times over all proofs. We call `parsing` the time required to parse and convert the veriT proof into Isabelle terms.

Overall, there are two kinds of rules: (1) direct application of a sequence of theorems—e.g., `equiv_pos2` corresponds to the theorem $\neg(a \Leftrightarrow b) \vee \neg a \vee b$—and (2) calls to full-blown tactics—like `qnt_cnf` (Sect. 4.2).

First, direct application of theorems are usually fast, but they occur so often that the cumulative time is substantial. For example, `cong` only needs to unfold

assumptions and apply reflexivity and symmetry of equality. However, it appears so often and sometimes on large terms, that it is an important rule.

Second, rules which require full-blown tactics are the slowest rules. For `qnt_cnf` (CNF under quantifiers, see Sect. 4.2), we have not written a specialized tactic, but rely on Isabelle's tableau-based `blast` tactic. This rule is rather slow, but is rarely used. It is similar to the rule `la_generic`: it is slow on average, but searching the coefficients takes even more time.

We can also see that the time required to check the simplification steps that were formerly combined into the `connect_equiv` rule is not significant anymore.

We have performed the same experiments with the reconstruction of the SMT solver Z3. In contrast to veriT, we do not have the amount of time required for parsing. The results are shown in Figs. 3 and 4. The rule distribution is very different. The `nnf-neg` and `nnf-pos` rules are the slowest rules and take a huge amount of time in the worst case. However, the coarser quantifier instantiation step is on average *faster* than the one produced by veriT. We suspect that reconstruction is faster because the rule, which is only an implication without choice terms, is easier to check (no equality reordering).

6 Related Work

The SMT solvers CVC4 [10], Z3 [34], and veriT [19] produce proofs. CVC4 does not record quantifier reasoning in the proof, and Z3 uses some macro rules. Proofs from SMT solvers have also been used to find unsatisfiability cores [20], and interpolants [32]. They are also useful to debug the solver itself, since unsound steps often point to the origin of bugs. Our work also relates to systems like Dedukti [5] that focuses on translating proof steps, not on replaying them.

Proof reconstruction has been implemented in various systems, including CVC4 proofs in HOL Light [31], Z3 in HOL4 and Isabelle/HOL [18], and veriT [4] and CVC4 [24] in Coq. Only veriT produces detailed proofs for preprocessing and skolemization. SMTCoq [4, 24] currently supports veriT's version 1 of the proof output which has different rules, does not support detailed skolemization rules, and is implemented in the 2016 version of veriT, which has worse performance. SMTCoq also supports bit vectors and arrays.

The reconstruction of Z3 proofs in HOL4 and Isabelle/HOL is one of the most advanced and well tested. It is regularly used by Isabelle users. The Z3 proof reconstruction succeeds in more than 90% of Sledgehammer benchmarks [14, Section 9] and is efficient (an older version of Z3 was used). Performance numbers are reported [16,18] not only for problems generated by proof assistants (including Isabelle), but also for preexisting SMT-LIB files from the SMT-LIB library.

The performance study by Böhme [16, Sect. 3.4] uses version 2.15 of Z3, whereas we use version 4.4.0 which currently ships with Isabelle. Since version 2.15, the proof format changed slightly (e.g., `th-lemma-arith` was introduced), fulfilling some of the wishes expressed by Böhme and Weber [18] to simplify reconstruction. Surprisingly, the `nnf` rules do not appear among the five rules that used the most runtime. Instead, the `th-lemma` and `rewrite` rules were the

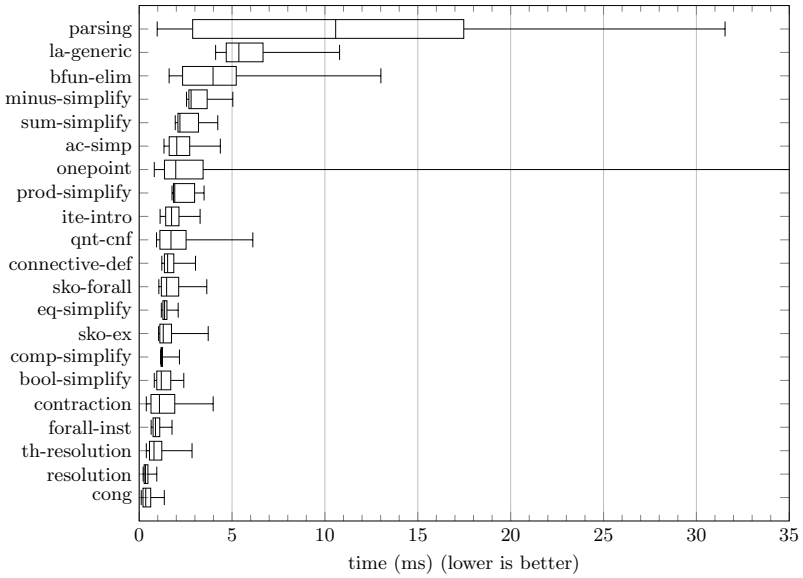


Fig. 1. Timing, sorted by the median, of a subset of veriT's rules. From left to right, the lower whisker marks the 5th percentile, the lower box line the first quartile, the middle of the box the median, the upper box line the third quartile, and the upper whisker the 95th percentile.

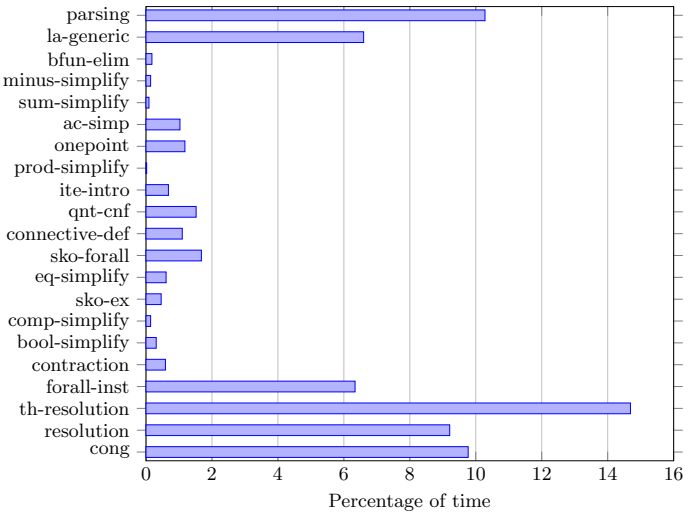


Fig. 2. Total percentage spent on each rule for the SMT solver veriT in the same order as Fig. 1. This graph maps the rules already shown in Fig. 1 to the total amount of time. The slowest rules are `th_resolution` (14.7%), parsing (10.3%), and `cong` (9.77%).

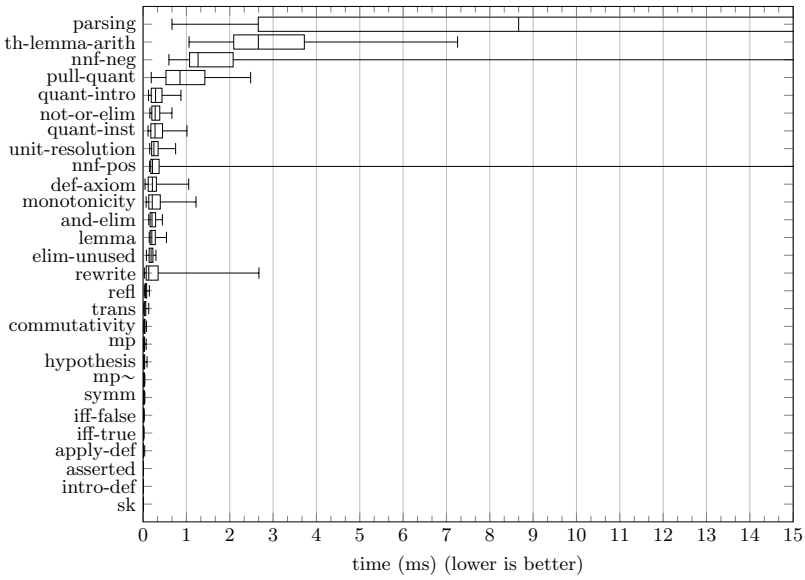


Fig. 3. Timing of some of Z3's rules sorted by median. From left to right, the lower whisker marks the 5th percentile, the lower box line the first quartile, the middle of the box the median, the upper box line the third quartile, and the upper whisker the 95th percentile. nnf-neg's 95th percentile is 87 ms, nnf-pos's is 33 ms, and parsing's is 25 ms.

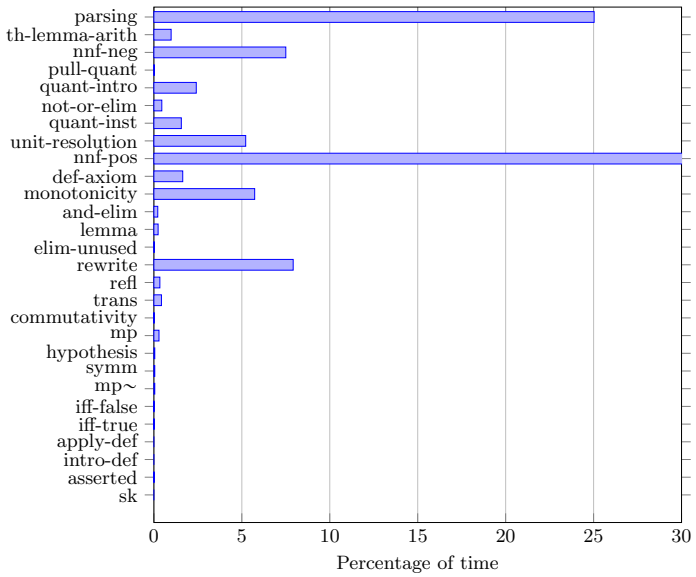


Fig. 4. Total amount of time per rule for the SMT solver Z3. nnf-neg takes 39% of the reconstruction time.

slowest. Similarly to veriT, the `cong` rule was among the most used (without accounting for the most time), but it does not appear in our Z3 tests.

CVC4 follows a different philosophy compared to veriT and Z3: it produces proofs in a logical framework with side conditions [39]. The output can contain programs to check certain rules. The proof format is flexible in some aspects and restrictive in others. Currently CVC4 does not generate proofs for quantifiers.

7 Conclusion

We presented an efficient reconstruction of proofs generated by a modern SMT solver in an interactive theorem prover. Our improvements address reconstruction challenges for proof steps of typical inferences performed by SMT solvers.

By studying the time required to replay each rule, we were able to compare the reconstruction for two different proof formats with different design directions. The very detailed proof format of veriT makes the reconstruction easier to implement and allows for more specialization of the tactics. On slow proofs, the ratio of time to reconstruct and time to find a proof is better for our more detailed format. Integrating our reconstruction in Isabelle halves the number of failures from Sledgehammer and nicely completes the existing reconstruction method with Z3.

Our work is integrated into Isabelle version 2021. Sledgehammer suggests the veriT-based reconstruction if it is the fastest tactic that finds the proof; so users profit without action required on their side. We plan to improve the reconstruction of the slowest rules and remove inconsistencies in the proof format. The developers of the SMT solver CVC4 are currently rewriting the proof generation and plan to support a similar proof format. We hope to be able to reuse the current reconstruction code by only adding support for CVC4-specific rules. Generating and reconstructing proofs from the veriT version with higher-order logic [9] could also improve the usefulness of veriT on Isabelle problems. The current proof rules [40] should accommodate the more expressive logic.

Acknowledgment We would like to thank Haniel Barbosa for his support with the implementation in veriT. We also thank Haniel Barbosa, Jasmin Blanchette, Pascal Fontaine, Daniela Kaufmann, Petar Vukmirović, and the anonymous reviewers for many fruitful discussions and suggesting many textual improvements. The first and third authors have received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreements No. 713999, Matryoshka, and No. 830927, Concordia). The second author is supported by the LIT AI Lab funded by the State of Upper Austria. The training presented in this paper was carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

1. Reliable Reconstruction of Fine-Grained Proofs in a Proof Assistant. Zenodo (Apr 2021). https://doi.org/10.5281/zenodo.4727349
2. Abdulaziz, M., Paulson, L.C.: An Isabelle/HOL formalisation of Green's theorem. Archive of Formal Proofs (Jan 2018), https://isa-afp.org/entries/Green.html, formal proof development
3. Abdulaziz, M., Paulson, L.C.: An Isabelle/HOL formalisation of Green's theorem. Journal of Automated Reasoning **63**(3), 763–786 (Nov 2019). https://doi.org/10.1007/s10817-018-9495-z
4. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of SAT/SMT solvers to Coq through proof witnesses. In: Jouannaud, J.P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 135–150. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_12
5. Assaf, A., Burel, G., Cauderlier, R., Delahaye, D., Dowek, G., Dubois, C., Gilbert, F., Halmagrand, P., Hermant, O., Saillard, R.: Expressing theories in the $\lambda\pi$-calculus modulo theory and in the Dedukti system. In: TYPES: Types for Proofs and Programs. Novi SAd, Serbia (May 2016)
6. Barbosa, H.: Efficient instantiation techniques in SMT (work in progress). vol. 1635, pp. 1–10. CEUR-WS.org (Jul 2016), http://ceur-ws.org/Vol-1635/#paper-01
7. Barbosa, H., Blanchette, J.C., Fleury, M., Fontaine, P.: Scalable fine-grained proofs for formula processing. Journal of Automated Reasoning (Jan 2019). https://doi.org/10.1007/s10817-018-09502-y
8. Barbosa, H., Fontaine, P., Reynolds, A.: Congruence closure with free variables. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 214–230. Springer Berlin Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_13
9. Barbosa, H., Reynolds, A., Ouraoui, D.E., Tinelli, C., Barrett, C.W.: Extending SMT solvers to higher-order logic. In: Fontaine, P. (ed.) CADE 27. LNCS, vol. 11716, pp. 35–54. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-29436-6_3
10. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
11. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at www.SMT-LIB.org
12. Barrett, C.W., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 305–343. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
13. Becker, H., Blanchette, J.C., Waldmann, U., Wand, D.: Formalization of Knuth–Bendix orders for lambda-free higher-order terms. Archive of Formal Proofs (Nov 2016), https://isa-afp.org/entries/Lambda_Free_KBOs.html, formal proof development
14. Blanchette, J.C., Böhme, S., Fleury, M., Smolka, S.J., Steckermeier, A.: Semi-intelligible Isar proofs from machine-generated proofs. Journal of Automated Reasoning **56**(2), 155–200 (2016). https://doi.org/10.1007/s10817-015-9335-3

15. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with smt solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 23. LNCS, vol. 6803, pp. 116–130. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_11
16. Böhme, S.: Proving Theorems of Higher-Order Logic with SMT Solvers. Ph.D. thesis, Technische Universität München (2012), http://mediatum.ub.tum.de/node?id=1084525
17. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. pp. 107–121. Springer Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_9
18. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 179–194. Springer Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_14
19. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) CADE 22. LNCS, vol. 5663, pp. 151–156. Springer Berlin Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_12
20. Déharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: SMT solvers for Rodin. In: Derrick, J., Fitzgerald, J.A., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 194–207. Springer Berlin Heidelberg (Jun 2012). https://doi.org/10.1007/978-3-642-30885-7_14
21. Dutertre, B., de Moura, L.: Integrating simplex with DPLL(T). Tech. rep., SRI International (May 2006), http://www.csl.sri.com/users/bruno/publis/sri-csl-06-01.pdf
22. Eberl, M.: Elementary facts about the distribution of primes. Archive of Formal Proofs (Feb 2019), https://isa-afp.org/entries/Prime_Distribution_Elementary.html, formal proof development
23. Eberl, M., Paulson, L.C.: The prime number theorem. Archive of Formal Proofs (Sep 2018), https://isa-afp.org/entries/Prime_Number_Theorem.html, formal proof development
24. Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., Barrett, C.W.: SMTCoq: A plug-in for integrating SMT solvers into Coq. In: Majumdar, R., Kuncak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 126–133. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-63390-9_7
25. Fleury, M., Schurr, H.: Reconstructing veriT proofs in Isabelle/HOL. In: Reis, G., Barbosa, H. (eds.) PxTP 2019. EPTCS, vol. 301, pp. 36–50 (2019). https://doi.org/10.4204/EPTCS.301.6
26. Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation, LNCS, vol. 78. Springer Berlin Heidelberg (1979). https://doi.org/10.1007/3-540-09724-4
27. Immler, F.: Re: [isabelle] Isabelle2019-RC2 sporadic smt failures. Email (May 2019), https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2019-May/msg00130.html
28. Kühlwein, D., Blanchette, J.C., Kaliszyk, C., Urban, J.: Mash: Machine learning for Sledgehammer. In: ITP. LNCS, vol. 7998, pp. 35–50. Springer (2013)
29. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 361–381. Springer International Publishing (2016). https://doi.org/10.1007/978-3-319-41528-4_20

30. Marić, F., Spasić, M., Thiemann, R.: An incremental simplex algorithm with unsatisfiable core generation. Archive of Formal Proofs (Aug 2018), https://isa-afp.org/entries/Simplex.html, formal proof development
31. McLaughlin, S., Barrett, C., Ge, Y.: Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. Electronic Notes in Theoretical Computer Science **144**(2), 43–51 (2006). https://doi.org/10.1016/j.entcs.2005.12.005
32. McMillan, K.L.: Interpolants from Z3 proofs. In: FMCAD 2011. pp. 19–27. FMCAD Inc, Austin, Texas (2011)
33. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Appl. Log. **7**(1), 41–57 (2009)
34. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer Berlin Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
35. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 112–131. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-89963-3_7
36. Reynolds, A., Tinelli, C., de Moura, L.: Finding conflicting instances of quantified formulas in SMT. In: FMCAD 2014. pp. 195–202. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987613
37. Schlichtkrull, A., Blanchette, J.C., Traytel, D., Waldmann, U.: Formalization of Bachmair and Ganzinger's ordered resolution prover. Archive of Formal Proofs (Jan 2018), https://isa-afp.org/entries/Ordered_Resolution_Prover.html, formal proof development
38. Schulz, S.: E - a brainiac theorem prover. AI Communications **15**(2-3), 111–126 (2002), http://content.iospress.com/articles/ai-communications/aic260
39. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Formal Methods in System Design **42**(1), 91–118 (Feb 2013). https://doi.org/10.1007/s10703-012-0163-3
40. The veriT Team and Contributors: Proofonomicon: A reference of the veriT proof format. Software Documentation (2021), https://www.verit-solver.org/documentation/proofonomicon.pdf, last Accessed: April 2021

Workshop Paper

[1] Hans-Jörg Schurr, Fleury, Mathias, Haniel Barbosa, and Pascal Fontaine. "Alethe: Towards a Generic SMT Proof Format (extended abstract)". In: *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, Pittsburg, USA, 11th July 2021*. Ed. by Chantal Keller and Fleury, Mathias. Vol. 336. Electronic Proceedings Ain Theoretical Computer Science. Open Publishing Association, 2021, pp. 49–54.

Alethe: Towards a Generic SMT Proof Format (extended abstract)

Hans-Jörg Schurr
University of Lorraine, CNRS, Inria, and LORIA, Nancy, France
hans-jorg.schurr@inria.fr

Mathias Fleury
Johannes Kepler University Linz, Austria
mathias.fleury@jku.at

Haniel Barbosa
Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
hbarbosa@dcc.ufmg.br

Pascal Fontaine
University of Liège, Belgium
pascal.fontaine@uliege.be

The first iteration of the proof format used by the SMT solver veriT was presented ten years ago at the first PxTP workshop. Since then the format has matured. veriT proofs are used within multiple applications, and other solvers generate proofs in the same format. We would now like to gather feedback from the community to guide future developments. Towards this, we review the history of the format, present our pragmatic approach to develop the format, and also discuss problems that might arise when other solvers use the format.

Over the years the production of machine-consumable formal proofs of unsatisfiability from SMT solvers [6] has attracted significant attention [4]. Such proofs enable users to certify unsatisfiability results similarly to how satisfiable results may be certified via models. However, a major difficulty that SMT proof formats must address is the complex and heterogeneous nature of SMT solvers: a SAT solver drives multiple, often very different, theory solvers; instantiation procedures generate ground instances; and heavily theory-dependent simplification techniques ensure that the solvers are fast in practice. Moreover, how each of these components works internally can also differ from solver to solver. As a testament to these challenges proof formats for SMT solvers have been mostly restricted to individual solvers and no standard has emerged. To be adopted by several solvers, an SMT proof format must be carefully designed to accommodate needs of specific solvers. This will require repeated refinement and generalization.

The basis for our efforts in this field is the proof format implemented by the SMT solver veriT [8] that is now mature and used by multiple systems. To further improve the format, as well as to accommodate not only the reasoning of the SMT solver veriT but also of other solvers, we are currently extending the format and developing better tooling, such as an independent proof checker. To facilitate this effort and overall usage, we are also writing a full specification. To emphasize the independence of the format we are baptizing it *Alethe*.[1] We do not presume to propose a standard format a priori. Instead we believe that Alethe, together with its tooling, can provide a basis for further discussions on how to achieve a format to be used by multiple solvers.

1 The State of Alethe

Alethe combines two major ideas whose roots reach back ten years to the first PxTP workshop in 2011 [7, 9]. It was proposed as an easy-to-produce format with a term language very close to SMT-LIB [5], the standard input language of SMT solvers, and rules with a varying level of granularity, allowing implicit

[1] Alethe is a genus of small birds and the name resembles *aletheia*, the Greek word for truth.

Chantal Keller and Mathias Fleury (Eds.): Seventh Workshop
on Proof eXchange for Theorem Proving (PxTP)
EPTCS 336, 2021, pp. 49–54, doi:10.4204/EPTCS.336.6

proof steps in the proof and thus relying on powerful proof checkers capable of filling the gaps. Since then the format has been refined and extended [2]. It is now mature, supports coarse- and fine-grained proof steps capturing SMT solving for the SMT-LIB logic UFLIRA[2] and can be reconstructed by the proof assistants Coq [1, 11] and Isabelle [12, 14]. In particular, the integration with Coq was also used as a bridge for the reconstruction of proofs from the SMT solver CVC4 [3] in Coq, where its proofs in the LFSC format [15] were first translated into the veriT format before reconstruction. Finally, the format will also be natively supported in the upcoming cvc5. solver[3]

On the one hand, Alethe uses a natural-deduction style calculus driven mostly by resolution [7]. To handle first-order reasoning, dedicated quantifier instantiation rules are used [9]. On the other hand, it implements novel ideas to express reasoning typically used for processing, such as Skolemization, renaming of variables, and other manipulations of bound variables [2]. While the format was always inspired by the SMT-LIB language, we recently [12] changed the syntax of Alethe to closely resemble the command structure used within SMT-LIB. When possible Alethe uses existing SMT-LIB features, such as the `define-fun` command to define constants and the `:named` annotation to implement term sharing.

The following proof fragment gives a taste of the format. The fragment first renames the bound variable in the term $\exists x.\, f(x)$ from x to vr and then skolemizes the quantifier. A proof is a list of commands. The *assume* command introduces an assumption, *anchor* starts a subproof, and *step* denotes an ordinary proof step. Steps are annotated with an identifier, a rule, and premises. The SMT-LIB command *define-fun* defines a function. The rule *bind* used by step `t1` performs the renaming of the bound variable. It uses a subproof (Steps `t1.t1` and `t1.t2`). The subproof uses a context to denote that `x` is equal to `vr` within the subproof. The `anchor` command starts the subproof and introduces the context. The `bind` rule does not only make it possible to rename bound variables, but within the subproof it is possible to simplify the formula as done during preprocessing. The steps `t2` and `t3` use resolution to finish the renaming. In step `t4` the bound variable is skolemized. Skolemization uses the choice binder ε and derives $f(\varepsilon vr.\, f(vr))$ from $\exists vr.\, f(vr)$. To simplify the reconstruction the choice term is introduced as a defined constant (by `define-fun`). Finally, resolution is used again to finish the proof.

```
(assume a0 (exists ((x A)) (f x)))
(anchor :step t1 :args (:= x vr))
(step t1.t1 (cl (= x vr)) :rule cong)
(step t1.t2 (cl (= (f x) (f vr))) :rule cong)
(step t1 (cl (= (exists ((x A)) (f x))
                (exists ((vr A)) (f vr)))) :rule bind)
(step t2 (cl (not (= (exists ((vr A)) (f x))
                     (exists ((vr A)) (f vr))))
             (not (exists ((vr A)) (f x)))
             (exists ((vr A)) (f vr))) :rule equiv_pos1)
(step t3 (cl (exists ((vr A)) (f vr))) :premises (a0 t1 t2) :rule resolution)
(define-fun X () A (choice ((vr A)) (f vr)))
(step t4 (cl (= (exists ((vr A)) (f vr)) (f X))) :rule sko_ex)
(step t5 (cl (not (= (exists ((vr A)) (f vr)) (f X)))
             (not (exists ((vr A)) (f vr)))
             (f X)) :rule equiv_pos1)
(step t6 (cl (f X)) :premises (t3 t4 t5) :rule resolution)
```

[2] That is the logic for problems containing a mix of any of quantifiers, uninterpreted functions, and linear arithmetic.
[3] `https://cvc4.github.io/2021/04/02/cvc5-announcement.html`

The output of Alethe proofs from veriT has now reached a certain level of maturity. The 2021 version of the Isabelle theorem prover was released earlier this year and supports the reconstruction of Alethe proofs generated by veriT. Users of Isabelle/HOL can invoke the `smt` tactic. This tactic encodes the current proof goal as an SMT-LIB problem and calls an SMT solver. Previously only the SMT solver Z3 was supported. Now veriT is supported too. If the solver produces a proof, the proof is reconstructed within the Isabelle kernel. In practice, users will seldom choose the `smt` tactic themselves. Instead, they call the Sledgehammer tool that calls external tools to find relevant facts. Sometimes, the external tool finds a proof, but the proof cannot be imported into Isabelle, requiring the user to write a proof manually. The addition of the veriT-powered `smt` tactic halves [14] the rate of this kind of failures. The improvement is especially pronounced for proofs found by CVC4. A key reason for this improvement is the support for the conflicting-instance instantiation technique within veriT. Z3, the singular SMT solver supported previously, does not implement this technique. Nevertheless, it is Alethe that allowed us to connect veriT to Isabelle, and we hope that the support for Alethe in other solvers will ease this connection between powerful SMT solvers and other tools in the future.

The process of implementing proof reconstruction in Isabelle also helped us to improve the proof format. We found both, possible improvements in the format (like providing the Farkas' coefficient for lemmas of linear arithmetic) and in the implementation (by identifying concrete errors). One major shortcoming of the proofs were rules that combined several simplification steps into one. We replaced these steps by multiple simple and well-defined rules. In particular every simplification rule addresses a specific theory instead of combining them. An interesting observation of the reconstruction in Isabelle is that some steps can be skipped to improve performance. For example, the proofs for the renaming of variables are irrelevant for Isabelle since this uses De Bruijn indices. This shows that reconstruction specific optimizations can counterbalance the proof length which is increased by fine-grained rules. We will take this prospect into account as we further refine the format.

2 A Glance Into the Future

The development of the Alethe proof format so far was not a monolithic process. Both practical considerations and research progress — such as supporting fine-grained preprocessing rules — influenced the development process. Due to this, the format is not fully homogeneous, but this approach allowed us to quickly adapt the format when necessary. We will continue this pragmatic approach.

Speculative Specification. We are writing a speculative specification.[4] During the development of the Isabelle reconstruction it became necessary to document the proof rules in a coherent and complete manner. When we started to develop the reconstruction there was only an automatically generated list of rules with a short comment for each rule. While this is enough for simple tautological rules, it does not provide a clear definition of the more complex rules such as the linear arithmetic rules. To rectify this, we studied veriT's source code and wrote an independent document with a list of all rules and a clear mathematical definition of each rule. We chose a level of precision for these descriptions that serves the implementer: precise enough to clarify the edge case, but without the details that would make it a fully formal specification. We are now extending this document to a full specification of the format. This specification is speculative in the sense that it will not be cast in stone. It will describe the format as it is in use at any point in time and will develop in parallel with practical support for the format within SMT solvers, proof checkers, and other tools.

[4] The current version is available at `http://www.verit-solver.org/documentation/alethe-spec.pdf`.

Flexible Rules. The next solver that will gain support for the Alethe format is the upcoming cvc5 solver. Implementing a proof format into another solver reveals where the proof format is too tied to the current implementation of veriT. On the one hand, new proof rules must be added to the format — e.g., veriT does not support the theory of bitvectors, while cvc5 does. When CVC4 was integrated into Coq via a translation of its LFSC proofs into Alethe proofs [11], an ad-hoc extension with bitvector rules was made. A revised version of this extension will now be incorporated into the upcoming specification of the format so that cvc5 bitvector proofs can be represented in Alethe. Further extensions to other theories supported by cvc5, like the theory of strings, will eventually be made as well.

Besides new theories, cvc5 can also be stricter than veriT in the usage of some rules. This strictness can simplify the reconstruction, since less search is required. A good example of this is the `trans` rule that expresses transitivity. This rule has a list of equalities as premises and the conclusion is an equality derived by transitivity. In principle, this rule can have three levels of "strictness":

1. The premises are ordered and the equalities are correctly oriented (like in cvc5), e.g., $a = b$, $b = c$, and $c = d$ implies $a = d$.
2. The premises are ordered but the equalities might not be correctly oriented (like in veriT), e.g., $b = a$, $c = b$, and $d = c$ implies $d = a$.
3. Neither are the assumptions ordered, nor are the equalities oriented, e.g., $c = b$, $b = a$, and $d = c$ implies $d = a$.

The most strict variant is the easiest to reconstruct: a straightforward linear traversal of the premises suffices for checking. From the point of view of producing it from the solver, however, this version is the hardest to implement. This is due to implementations of the congruence closure decision procedure [13, 10] in SMT solvers being generally agnostic to the order of equalities, which can lead to implicit reorientations that can be difficult to track. Anecdotally, for cvc5 to achieve this level of detail several months of work were necessary, within the overall effort of redesigning from scratch CVC4's proof infrastructure. Since we cannot assume every solver developer will, or even should, undertake such an effort, all the different levels of granularity must be allowed by the format, each requiring different complexity levels of checking.

To keep the proof format flexible and proofs easy to produce, we will provide different versions of proof rules, with varying levels of granularity as in the transitivity example case above, by *annotating* them. This leverages the rule *arguments*, which are already used by some rules. For example, the Farkas' coefficient of the linear arithmetic rule are provided as arguments. This puts pressure on proof checkers and reconstruction in proof assistants to support all the variants or at least the most general one (at the cost of efficiency). Hence, our design principle here is that the annotation is optional: the absence of an annotation denotes the least strict version of the rule.

Powerful Tooling. We believe that powerful software tools may greatly increase the utility of a proof format. Towards this end we have started implementing an independent proof checker for Alethe. In contrast to a proof-assistant-based reconstruction, this checker will not be structured around a small, trusted kernel, and correct-by-construction extensions. Instead, the user would need to trust the implementation does not lead to wrong checking results. Instead, its focus is on performance, support for multiple features and greater flexibility for integrating extensions and refinements to the format. The Isabelle checker is currently not suited to this task — one major issue is that it does not support SMT-LIB input files.[5]

[5] A version capable of doing so was developed for Z3 but it was unfortunately lost.

This independent checker will also serve as a proof "elaborator". Rather than checking, it will also allow converting a coarse-grained proof, containing implicit steps, to a fine-grained one, with more detailed steps. The resulting proof can then be more efficiently checked by the tool itself or via proof-assistant reconstructions. An example of such elaboration is the transitivity rule. If the rule is not in its most detailed version, with premises in the correct order and none implicitly reordered, it can be elaborated by greedily reordering the premises and adding proof steps using the symmetry of equality. Note however that in the limit detailing coarse-grained steps can be as hard as solving an SMT problem. Should such cases arise, the checker will rely on internal proof-producing procedures capable of producing a detailed proof for the given step. At first the veriT and cvc5 solvers, which can produce fine-grained proofs for several aspects of SMT solving, could be used in such cases.

A nice side effect of the use of an external checker is that it could prune useless steps. Currently SMT solvers keep a full proof trace in memory and print a pruned proof after solving finishes. This is in contrast to SAT solvers that dump proofs on-the-fly. For SAT proofs, the pruned proof can be obtained from a full trace by using a tool like DRAT-TRIM. There is some ongoing work by Nikolaj Bjørner on Z3 to also generate proofs on-the-fly, but it is not clear how to support preprocessing and quantifiers.[6]

3 Conclusion

We have presented on overview of the current state of the Alethe proof format and some ideas on how we intend to improve and extend the format, as well as supporting tools. In designing a new proof format supported across two solvers we hope to provide a first step towards a format adopted by more solvers. This format allows several levels of detail, and is thus flexible enough to reasonably easily produce proofs in various contexts. We intend to define a precise semantics at each level though. This distinguishes our format from other approaches, such as the TSTP format [16], that are probably easier to adopt but only specify the syntax, leading to very different proofs generated by the various provers supporting it.

One limit of our approach for proofs is that we cannot express global transformations like symmetry breaking. SAT solvers are able to add clauses (DRAT clauses) such that the overall problems is equisatisfiable. It is unclear however how to add such clauses in the SMT context.

Overall, we hope to get feedback from users and developers to see what special needs they have and exchange ideas on the proof format.

Acknowledgment We thank Bruno Andreotti for ongoing work on the proof checker and Hanna Lachnitt for ongoing work on the cvc5 support. We are grateful for the helpful comments provided to us by the anonymous reviewers. The first author has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). The second author is supported by the LIT AI Lab funded by the State of Upper Austria.

References

[1] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In Jean-Pierre Jouannaud & Zhong Shao, editors: *Certified Programs and Proofs*, *LNCS* 7086, Springer, pp. 135–150, doi:10.1007/978-3-642-25379-9_12.

[6] `https://github.com/Z3Prover/z3/discussions/4881`

[2] Haniel Barbosa, Jasmin C. Blanchette, Mathias Fleury & Pascal Fontaine (2019): *Scalable Fine-Grained Proofs for Formula Processing*. Journal of Automated Reasoning, doi:10.1007/s10817-018-09502-y.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds & Cesare Tinelli (2011): *CVC4*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: Computer Aided Verification (CAV), Springer, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.

[4] Clark Barrett, Leonardo De Moura & Pascal Fontaine (2015): *Proofs in satisfiability modulo theories*. All about proofs, Proofs for all 55(1), pp. 23–44.

[5] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2017): *The SMT-LIB Standard: Version 2.6*. Technical Report, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[6] Clark W. Barrett & Cesare Tinelli (2018): *Satisfiability Modulo Theories*. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith & Roderick Bloem, editors: Handbook of Model Checking., Springer, pp. 305–343, doi:10.1007/978-3-319-10575-8_11.

[7] Frédéric Besson, Pascal Fontaine & Laurent Théry (2011): *A Flexible Proof Format for SMT: A Proposal*. In Pascal Fontaine & Aaron Stump, editors: *PxTP 2011*, pp. 15–26.

[8] Thomas Bouton, Diego C. B. de Oliveira, David Déharbe & Pascal Fontaine (2009): *veriT: An Open, Trustable and Efficient SMT-solver*. In Renate A. Schmidt, editor: *CADE 22*, *LNCS* 5663, Springer Berlin Heidelberg, pp. 151–156, doi:10.1007/978-3-642-02959-2_12.

[9] David Déharbe, Pascal Fontaine & Bruno Woltzenlogel Paleo (2011): *Quantifier Inference Rules for SMT Proofs*. In Pascal Fontaine & Aaron Stump, editors: *PxTP 2011*, pp. 33–39.

[10] Peter J. Downey, Ravi Sethi & Robert Endre Tarjan (1980): *Variations on the Common Subexpression Problem*. *J. ACM* 27(4), pp. 758–771, doi:10.1145/322217.322228.

[11] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds & Clark Barrett (2017): *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*. In Rupak Majumdar & Viktor Kunčak, editors: *CAV 2017*, *LNCS* 10426, Springer, pp. 126–133, doi:10.1007/978-3-319-63390-9_7.

[12] Mathias Fleury & Hans-Jörg Schurr (2019): *Reconstructing veriT Proofs in Isabelle/HOL*. In Giselle Reis & Haniel Barbosa, editors: *PxTP 2019*, *EPTCS* 301, pp. 36–50, doi:10.4204/EPTCS.301.6.

[13] Greg Nelson & Derek C. Oppen (1980): *Fast Decision Procedures Based on Congruence Closure*. *J. ACM* 27(2), pp. 356–364, doi:10.1145/322186.322198.

[14] Hans-Jörg Schurr, Mathias Fleury & Martin Desharnais (2021): *Reliable Reconstruction of Fine-Grained Proofs in a Proof Assistant*. In: *CADE 28*, LNCS.

[15] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean & Cesare Tinelli (2013): *SMT proof checking using a logical framework*. Formal Methods in System Design 42(1), pp. 91–118, doi:10.1007/s10703-012-0163-3.

[16] Geoff Sutcliffe, Jürgen Zimmer & Stephan Schulz (2004): *TSTP Data-Exchange Formats for Automated Theorem Proving Tools*. In Weixiong Zhang & Volker Sorge, editors: Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems, Frontiers in Artificial Intelligence and Applications 112, IOS Press, pp. 201–215.

Workshop Paper

[1] Hanna Lachnitt, Mathias Fleury, Leni Aniva, Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. "IsaRare: Automatic Verification of SMT Rewrites in Isabelle/HOL". In: *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*. Ed. by Bernd Finkbeiner and Laura Kovács. Vol. 14570. Lecture Notes in Computer Science. Accepted at TACAS 2024, I supervised Hanna on her work in Isabelle. Springer, 2024, pp. 311–330.

IsaRare: Automatic Verification of SMT Rewrites in Isabelle/HOL *

Hanna Lachnitt[1], Mathias Fleury[4], Leni Aniva[1], Andrew Reynolds[2], Haniel Barbosa[3], Andres Nötzli[5], Clark Barrett[1], and Cesare Tinelli[2]

[1] Stanford University, Stanford, USA
[2] The University of Iowa, Iowa City, USA
[3] Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
[4] University of Freiburg, Freiburg, Germany
[5] Cubist, Inc.

Abstract. Satisfiability modulo theories (SMT) solvers are widely used to ensure the correctness of safety- and security-critical applications. Therefore, being able to trust a solver's results is crucial. One way to increase trust is to generate independently checkable proof certificates, which record the reasoning steps done by the solver. A key challenge with this approach is that it is difficult to efficiently and accurately produce proofs for reasoning steps involving term rewriting rules. Previous work showed how a domain-specific language, Rare, can be used to capture rewriting rules for the purposes of proof production. However, in that work, the Rare rules had to be trusted, as the correctness of the rules themselves was not checked by the proof checker. In this paper, we present IsaRare, a tool that can automatically translate Rare rules into Isabelle/HOL lemmas. The soundness of the rules can then be verified by proving the lemmas. Because an incorrect rule can put the entire soundness of a proof system in jeopardy, our solution closes an important gap in the trustworthiness of SMT proof certificates. The same tool also provides a necessary component for enabling full proof reconstruction of SMT proof certificates in Isabelle/HOL. We evaluate our approach by verifying an extensive set of rewrite rules used by the cvc5 SMT solver.

1 Introduction

Satisfiability modulo theories (SMT) [8] solvers provide the back-end reasoning power for many formal methods applications. These applications are often used to provide safety or security guarantees for critical systems [1, 15, 21, 23]. For such applications, an incorrect result from a solver could have catastrophic consequences. Thus, ensuring the correctness of a solver's results is crucial. However, industrial-strength SMT solvers are large and complex software systems which are under constant active development. As with any other large software project,

* This work was supported in part by the Stanford Center for Automated Reasoning and by a gift from Amazon Web Services.

even when employing software engineering best practices, it is unrealistic to expect that solvers do not contain implementation bugs that could, in the worst case, compromise the correctness of their answers.

One solution is to formally verify the SMT solver itself. Unfortunately, that would be a massive effort. It would likely require performance compromises [17] and impose a tremendous maintenance burden, as changes to solvers are frequent, and each change would require revisiting the verification.

Fortunately, there is a less expensive alternative: we can independently check each result produced by a solver. This is generally easy when the result is "satisfiable," at least for quantifier-free inputs. The solver can produce a model and we can check via evaluation that the input formula indeed holds in it. To have a similar ability to check a result of "unsatisfiable," solvers must be instrumented to produce *proof certificates* that can be independently verified by a separate proof checker. To maximize trustworthiness, the proof checker should be small, simple, and, ideally, formally verified. Alternatively, the checker can be embedded in a highly trusted system such as a skeptical interactive theorem prover. The SMT community is increasingly embracing proof production, with it becoming a major focus in recent years [3, 4, 19, 29].

One of the main challenges faced by SMT proof production efforts is the extensive use of theory-specific *term rewriting rules*. There are hundreds of such rules in modern solvers, each of which must be justifiable using some proof rule. Nötzli et al. [28] introduced a methodology for producing proofs for term rewriting rules by using the Rare domain-specific language. In that work, rules are defined in Rare, imported by a solver, and then used to elaborate the solver's term rewriting proof steps into finer-grained proofs using the Rare rules. This approach has proved to be viable in the cvc5 SMT solver [2]. However, previous work did not address the *correctness* of the rules, i.e., it does not ensure that an incorrect Rare rule does not compromise the correctness of proof certificates.

An incorrect rule can have severe consequences. First of all, it may affect the ability of the solver to produce a proof certificate at all: if the incorrect rule does not match what the solver code does, then the elaboration of the term rewriting proof steps with Rare may fail. More concerningly, if both the code and the proof rule are incorrect in the same way (perhaps because one was modeled after the other), then proof elaboration may succeed, but the proof certificate will be incorrect because it uses an invalid rule. This is especially problematic when using proof checkers that consider proof rules as trusted—that is, they only check whether rules are applied correctly and do not check the rules themselves.

There are two ways to fill this gap. One is to separately verify the proof rules; another is to use a more sophisticated proof checker, for example, one embedded in a skeptical interactive theorem prover, that will fail if an invalid rule is used. In this paper, we introduce IsaRare, a new plugin for the Isabelle/HOL proof assistant [27] (abbreviated to just Isabelle going forward), which can do the former and is a necessary step towards the latter. The plugin translates Rare rules into the language of Isabelle where they can then be formally proved as lemmas. Note that when using IsaRare simply as a rewrite rule verifier, the translation

from Rare to Isabelle becomes another trusted component. We mitigate this by reusing extensively-tested infrastructure in Isabelle for the translation.

To show the effectiveness of IsaRare, we implemented a large number of new rules in Rare (beyond those in [28]) needed to elaborate term rewriting steps in proofs generated by the cvc5 SMT solver [2]. We show that IsaRare can translate all of these rules into corresponding lemmas in Isabelle and can prove the majority of them automatically. In ongoing work, we are manually providing proofs for the rest, and have already proven most of them.

Our long-term vision is to enable the full integration of cvc5 and Isabelle via proof certificate reconstruction. Currently, Isabelle can send proof obligations to cvc5, but it is unable to automatically reconstruct Isabelle proofs from cvc5's proof certificates. Our goal is to enable Isabelle to reconstruct *every* step in these proof certificates. In order to reach this goal, it is essential to have rewrite lemmas available for reconstructing rewrite steps, as they appear in almost all proofs, and without dedicated support for discharging rewrite proof steps, reconstruction in Isabelle can fail [11, 31].

In summary, we make the following contributions:

- we introduce IsaRare, an Isabelle plugin for generating correctness lemmas for Rare rules;
- we add several new features to Rare itself and implement 163 new rewrite rules in Rare, almost tripling the size of the rule database from [28];
- we evaluate IsaRare, showing that it can translate all of the Rare rules into Isabelle lemmas and can prove the majority of them automatically.

In the rest of the paper, after surveying related work, we give an overview of proof production and the interface to Isabelle (Section 2). Then, we present the Rare language and our extensions (Section 3). We next introduce IsaRare and explain the challenges in transforming a Rare rule to an Isabelle lemma (Section 4). Finally, we present an evaluation of our approach (Section 5).

1.1 Related Work

Various attempts at proof production in SMT solvers have been implemented in the past [7, 13, 14, 22, 25], though these implementations typically either produce proofs certificates that are too coarse-grained (that is, they do not provide enough information for efficient proof checking) or produce them only if critical components are disabled, making solving while producing proofs slow or incomplete. Producing complete, independently-checkable proofs remains challenging.

One major challenge is solved by the modular framework by Barbosa et al. [3]. It enables proof production during term rewriting and formula processing and has been implemented in the SMT solver veriT [13] using the Alethe proof format [32]. Hoenicke and Schindler [19] introduce an alternative approach, implemented in the solver SMTInterpol [14], which also allows proof production for term rewriting and formula processing. Both of these approaches assume that the set of rewrite rules that can be used in proofs is fixed. Their sets include rules

for rewriting over equality, rules for rewriting Boolean formulas, and rules for reasoning about arithmetic. Notably absent, however, are rules for string and bit-vector rewrites. In other work, Barbosa et al. [4] describe a general architecture where the only holes in the generated proof certificates are those from rewrite steps. One of their key ideas is to support lazy proof production via a post-processing proof reconstruction step. This capability is leveraged in the work by Nötzli et al. [28] to produce proofs for rewrite steps based on rules written in Rare, which is the starting point for this work.

The interactive theorem prover Isabelle [30] includes a popular tool called Sledgehammer [9], which encodes proof obligations as SMT problems and uses SMT solvers to solve them. Sledgehammer currently supports proof reconstruction [12, 18] for two SMT solvers: z3 [26] and veriT [13]. However, z3 provides only coarse-grained proofs, which can cause reconstruction to fail. This issue was addressed for veriT by manually translating and proving correct in Isabelle the predefined set of rewrite rules in Alethe [18, 31]. Our work improves on this effort by providing an *automatic* mechanism for translating an *extendable* set of rewrite rules into Isabelle and includes support for bit-vector and string rewrites unsupported by veriT.

2 Preliminaries

2.1 Satisfiability Modulo Theories (SMT)

The underlying logic of SMT is many-sorted first-order logic with equality (see e.g., [16]). A *signature* Σ consists of a set $\Sigma^{\mathrm{s}} \subseteq S$ of sort symbols and a set Σ^{f} of sorted function symbols with sorts from Σ^{s}. We assume the usual definitions of well-sorted terms, literals, and formulas. We also use the usual definition of interpretations and of a satisfiability relation $\models$ between Σ-interpretations and Σ-formulas. A *Σ-theory* T is a non-empty class of Σ-interpretations closed under variable reassignment. A Σ-formula φ is *T-satisfiable* (resp., *T-unsatisfiable*, *T-valid*) if it is satisfied by some (resp., no, all) interpretation(s) in T. For the rest of the paper, we assume (un)satisfiability is always with respect to some given background theory T.

2.2 SMT Proofs and Rewriting

A *proof (of unsatisfiability)* is a series of inference steps starting from an input formula and terminating with $\bot$, showing that the input formula is unsatisfiable. The *granularity* of a proof step refers to how much reasoning it requires and roughly corresponds to the complexity of checking that the step is correct. In particular, steps (and thus the proofs containing them) are *fine-grained* if they can be efficiently checked, and *coarse-grained* otherwise. We will often refer to coarse-grained steps as *holes*.

One approach for the efficient production of proofs is to introduce coarse-grained proof steps for certain performance-critical deductions made while solv-

ing and then go back and fill in these holes with fine-grained steps as a post-processing step. We refer to this as *proof elaboration*, and it is particularly appealing for rewriting steps, since SMT solvers have hundreds of different rewrites to simplify and normalize terms, and instrumenting the rewriting code to produce fine-grained proofs is difficult and may introduce an unacceptable degradation in performance.

The approach taken by Nötzli et al. [28], and the one we also follow in this paper, is to assume that the SMT solver uses *generic* proof steps for all rewrites during solving and then elaborates these steps during post-processing by consulting a database of specific rewrite rules. The database is constructed by defining a set of rewrite rules in the domain-specific language Rare, which we discuss in Section 3. The elaboration tries to find one or more rules from the database to justify each generic, coarse-grained rewrite step. Additionally, it uses a built-in *evaluate* rule to justify steps that hold purely via constant folding. If elaboration is successful, the generic step is replaced by the fine-grained steps from the database.

2.3 SMT in Isabelle

As mentioned above, Sledgehammer [9] is an Isabelle tactic that applies automated reasoning tools, including SMT solvers, to prove goals in Isabelle. When targeting an SMT solver, the goal is encoded as an SMT-LIB [5] problem which is unsatisfiable iff the goal is valid. Sledgehammer also selects facts that it thinks will be relevant for solving the goal and includes encodings of them as well. The problem is given to the solver which reports back to Sledgehammer whether it was able to prove the goal [9]. Proving the goal externally, however, is not enough since Isabelle is a *skeptical* proof assistant, in the sense that it does not trust external solvers. Thus, a proof of the goal must somehow be constructed and checked inside Isabelle.

Finding such a proof internally can be challenging. One useful technique is to query the external solver for an *unsat core*, i.e., a subset of the facts it was given that are sufficient to prove the goal valid. Sometimes, this information is enough for Isabelle to search for an internal proof on its own. However, this process can be greatly improved, if, instead of just communicating the result and the core back to Sledgehammer, the solver also communicates a fine-grained proof. Then, with the appropriate proof reconstruction machinery, each step in the proof can be reconstructed as one or more steps using Isabelle's internal inference engine. As mentioned in Section 1.1, Sledgehammer can do this for proofs from the veriT and z3 solvers, though the former supports only a limited set of theories, and the latter produces only coarse-grained proofs.

Still, this means that Isabelle already has an integration with solvers supporting the SMT-LIB standard and is able to translate to and from SMT-LIB and internal terms. We build on this integration and extend it. Notice that such an integration requires each SMT-LIB operator to be matched with a term in Isabelle with the same semantics. Isabelle has built-in operators that match well with those in the uninterpreted function and arithmetic SMT theories, and both

formalisms support quantifiers [18]. However, Isabelle only has partial support for bit-vector operators. A more complete development of bit-vectors in Isabelle is described by Böhme et al. [11], but unfortunately, parts of their work (including parsing bit-vector proofs) never made it into Isabelle and now appear to be lost. As we describe below, part of our effort includes improving support for SMT theories in Isabelle, including bit-vectors and strings.

2.4 Approximate Sorts

Rare rules are meant to be easy and effortless to write. This is not the case when users have to specify sort information that is either inferable from the context or too restrictive. As an example of the latter, consider any rewrite rules involving bit-vector sorts. The SMT-LIB standard provides bit-vectors sorts that are parameterized by their size, or *bit-width*. However, to keep sort checking simple, it requires all bit-widths in SMT-LIB scripts to be concrete as, for instance, in `(_ BitVec 8)`. A similar argument applies to polymorphic sorts because, although SMT-LIB allows the definition of theories with such sorts (such as, for instance, array, set, and sequence sorts), it restricts scripts to monomorphic instantiations of polymorphic sorts — e.g., `(Set Int)`.

Unfortunately, these restrictions are too strong for Rare. They make it impossible, for example, to write any rewrite rule involving bit-vector terms that is naturally parametric in the bit-width of those terms, or any rule involving terms with a polymorphic sort. The ideal solution would then be to introduce dependent types (or sorts, to maintain the SMT-LIB terminology) in Rare, allowing both value and type parameters in sorts — e.g., `(_ BitVec n)` with `n` an integer variable, and `(Array A B)` with `A` and `B` type variables. However, this would make it difficult for SMT solvers, cvc5 included, to process Rare rules since, effectively, they only support non-dependent, monomorphic sorts.

Rare's compromise solution is to add instead *approximate sorts* to the sort system, following an approach analogous to gradual typing in programming languages [33], a hybrid type-checking discipline where some program types are checked statically and others are checked dynamically. In our case, where there is no notion of dynamic checking, we have instead two sort-checking phases in the SMT solver for Rare rules: (*i*) as the rules are read by the solver, when sort checking is done with respect to the declared approximate sorts, and (*ii*) during proof elaboration, when the approximate sorts in the Rare rules are matched against the exact sorts in the proof steps that correspond to those rules.

Approximate sorts are obtained by extending the sort system of SMT-LIB with a distinguished unknown value and a distinguished unknown sort, both denoted by `?`, that can be used in place of a value or parameter in a sort. This allows the construction of approximate sorts such as `(_ BitVec ?)`, `(Set ?)`, and `(Array ? ?)` (abbreviated as `?BitVec`, `?Set`, and `?Array`), while still allowing precise sorts such as `(_ BitVec 1)`, `(Set Real)`, and `(Array Int Real)`. Approximate sorts can be used to approximate dependently-sorted/polymorphic rewrite rules, as we see in the next section.

$$
\begin{array}{lll}
\langle rule \rangle & ::= & \texttt{(}\ \texttt{define-rule}\ \langle symbol \rangle\ \texttt{(}\ \langle par \rangle^*\ \texttt{)}\ [\langle defs \rangle]\ \langle expr \rangle\ \langle expr \rangle\ \texttt{)} \\
& | & \texttt{(}\ \texttt{define-rule*}\ \langle symbol \rangle\ \texttt{(}\ \langle par \rangle^*\ \texttt{)}\ [\langle defs \rangle]\ \langle expr \rangle\ \langle expr \rangle\ [\langle expr \rangle]\ \texttt{)} \\
& | & \texttt{(}\ \texttt{define-cond-rule}\ \langle symbol \rangle\ \texttt{(}\ \langle par \rangle^*\ \texttt{)}\ [\langle defs \rangle]\ \langle expr \rangle\ \langle expr \rangle\ \langle expr \rangle\ \texttt{)} \\
\langle par \rangle & ::= & \langle symbol \rangle\ \langle sort \rangle\ [\texttt{:list}] \\
\langle sort \rangle & ::= & \langle symbol \rangle \mid \texttt{?} \mid \texttt{?}\langle symbol \rangle \mid \texttt{(}\ \langle symbol \rangle\ \langle numeral \rangle^+\ \texttt{)} \\
\langle expr \rangle & ::= & \langle const \rangle \mid \langle id \rangle \mid \texttt{(}\ \langle id \rangle\ \langle expr \rangle^+\texttt{)} \\
\langle id \rangle & ::= & \langle symbol \rangle \mid \texttt{(}\ \langle symbol \rangle\ \langle numeral \rangle^+\ \texttt{)} \\
\langle binding \rangle & ::= & \texttt{(}\ \langle symbol \rangle\ \langle expr \rangle\ \texttt{)} \\
\langle defs \rangle & ::= & \texttt{(}\ \texttt{def}\ \langle binding \rangle^+\ \texttt{)}
\end{array}
$$

Fig. 1: Overview of the grammar of Rare.

An additional advantage of this approach is that, by relieving the Rare user from the burden of specifying the precise sort of variables in rewrite rules, it makes them both easier to write and less error-prone. At the same time, the loss of precision introduced by approximate sorts is not a serious hindrance in practice: both the SMT solver, which relies on Rare rules for proof elaboration, and IsaRare, which uses them during proof reconstruction, are able to infer the exact sort represented by an approximate one thanks to their knowledge of the (exact) sort of the constant and function symbols in the supported SMT theories. Subsection 4.3 explains how IsaRare recovers exact sorts by type inference fully automatically during the translation to Isabelle.

3 The RARE Language

The Rare language[6] was introduced by Nötzli et al. [28]. As part of this work, we have extended the language to be able to represent more rewrite rules. We present the full updated language here and summarize the differences with [28] at the end of the section.

A Rare file contains a list of rules whose syntax is defined by the grammar in Figure 1. Expressions use SMT-LIB syntax with a few exceptions. These include the use of approximate sorts for parameterized sorts (e.g., arrays and bit-vectors) and the addition of a few extra operators (e.g., `bvsize`, described below). Rare uses SMT-LIB 3 syntax [6], which is very close to SMT-LIB 2 and mostly differs from its predecessor in that it uses higher-order functions for indexed operators.

We say that an expression e *matches* a match expression m if there is some *matching substitution* σ that replaces each variable in m by a term of the same sort to obtain e (i.e., $m\sigma$ is syntactically identical to e). For example, the expression `(or (bvugt x1 x2) (= x2 x3))`, with variables `x1`, `x2`, `x3`, all of sort `?BitVec`,

[6] Rare comes from Rewrites, Automatically reconstructed.

matches `(or (bvugt a b) (= b a))` but not `(or (bvugt a b) (= c a))`, with `a`, `b`, and `c` bit-vector constant symbols of the same bit-width.

Rare Rules A Rare rewrite rule is defined with the `define-rule` command which starts with a parameter list containing variables with their sorts. These variables are used for matching as explained below. After an optional *definition list* (see below), there follow two expressions that form the main body of the rule: the *match* expression and the *target* expression. The semantics of a rule with match expression m and target expression t is that any expression e matching m under some sort-preserving matching substitution σ can be replaced by $t\sigma$. With approximate sorts, the sort preservation requirement is relaxed as follows. In Rare, for any sort constructor S of arity $n > 0$, there is a corresponding approximate sort `(`S `?` $\cdots$ `?)` with n occurrences of `?` which is always abbreviated as `?`S. A variable x with sort `?`S (e.g., `?BitVec`) in a match expression matches all expressions whose sort is constructed with S (e.g., `(BitVec 1)`, `(BitVec 2)`, and so on). Variables with sort `?` match expressions of any sort.

An optional *definition list* may appear in a Rare rule immediately after the parameter list. It starts with the keyword `def` and provides a list of local variables and their definitions, allowing the rewrite rule to be expressed more succinctly. A rule with a definition list is equivalent to the same rule without it, where each variable in the definition list has been replaced by its corresponding expression in the body of the rule. For a rule to be well-formed, all variables in the match and target expressions must appear either in the parameter list or the definition list. Similarly, each variable in the parameter list must appear in the match expression (while this second requirement could be relaxed, it is useful for catching mistakes). Consider the following example.

```
(define-rule bv-sign-extend-eliminate ((x ?BitVec) (n Int))
  (def (s (bvsize x)))
  (sign_extend n x)   (concat (repeat n (extract (- s 1) (- s 1) x)) x))
```

In this rule, there are two parameters, `x` and `n`. The sort annotation `?BitVec` indicates that `x` is a bit-vector without specifying its bit-width. The latter is stored in the local variable `s` using the `bvsize` operator. The rule says that a `(sign_extend n x)` expression can be replaced by repeating `n` times the most significant bit of `x` and then prepending this to `x`.

The `define-cond-rule` command is similar to `define-rule` except that it has an additional expression, the *condition*, immediately after the parameter and definition lists. This restricts the rule's applicability to cases where the condition can be proven equivalent to true under the matching substitution. In the example below, the condition `(> n 1)` can be verified by evaluation since in SMT-LIB, the first argument of `repeat` must be a numeral.

```
(define-cond-rule bv-repeat-eliminate-1 ((x ?BitVec) (n Int))
  (> n 1)   (repeat n x)   (concat x (repeat (- n 1) x)))
```

Note that the rule does not apply to terms like `(repeat 1` t`)` or `(repeat 0` t`)`.

Fixed-point Rules The `define-rule*` command defines rules that should be applied repeatedly, to completion. This is useful, for instance, in writing rules that iterate over the arguments of n-ary operators. Its basic form, with a body containing just a match and target expression, defines a rule that, whenever is applied, must be applied again on the resulting term until it no longer applies.

The user can optionally supply a *context* to control the iteration. This is a third expression that must contain an underscore. The semantics is that the match expression rewrites to the context expression, with the underscore replaced by the target expression. Then the rule is applied again to the target expression only. In the example below, the `:list` modifier is used to represent an arbitrary number of arguments, including zero, of the same type.

```
(define-rule* bv-neg-add ((x ?BitVec) (y ?BitVec) (zs ?BitVec :list))
  (bvneg (bvadd x y zs))   (bvneg (bvadd y zs))   (bvadd (bvneg x) _))
```

This rule rewrites a term `(bvneg (bvadd` s t $\cdots$`))` to the term `(bvadd (bvneg` s`)` r`)` where r is the result of recursively applying the rule to `(bvneg (bvadd` t $\cdots$`))`.

Changes to Rare Here, we briefly mention the changes to Rare with respect to [28]. First, we have support for a richer class of approximate sorts, including approximate bit-vector and array sorts. Also, we replaced the `let` construct by the new `def` construct. The definition list is more powerful as it applies to the entire rest of the body (whereas `let` was local to a single expression).

Additionally, to aid with bit-vector rewrite rules, we added several operators: `bvsize`, which returns the width of an expression of sort `?BitVec`; `bv`, which takes a integer n and natural w, and returns a bit-vector of width w and value $n \bmod 2^w$; `int.log2` which returns the integer (base 2) logarithm of an integer, and `int.islog2`, which returns true iff its integer argument is a power of 2.

We also removed the `:const` modifier, which was used previously to indicate that a particular expression had to be a constant value. We found that this adds complexity and is usually unnecessary. For rules that actually manipulate specific constant values, we can specify those values explicitly, e.g., by using the `bv` operator above.

4 IsaRare: from Rare Rewrites to Isabelle Lemmas

In this section, we introduce IsaRare, a plugin for Isabelle that automatically translates a Rare rule into an Isabelle lemma stating the correctness of the rule. Being able to generate such lemmas automatically is highly desirable, as Rare rules may be added and/or changed frequently for a given solver, or differ significantly between solvers, and manually translating Rare rules into lemmas is time-consuming and error-prone. IsaRare can also suggest a proof sketch which is sometimes sufficient to prove the lemma. If this automatic proof fails, the user must provide the proof or determine that the lemma does not hold. In the latter case, Isabelle's counterexample-finder Nitpick [10] can be helpful.

```
(define-cond-rule str-len-replace-inv ((t String) (s String) (r String))
  (= (str.len s) (str.len r))
  (str.len (str.replace t s r))   (str.len t))
```

```
lemma str_len_replace_inv:
  fixes t::string and s::string and r::string
  shows "smtlib_str_len s = smtlib_str_len r ⟶
   smtlib_str_len (smtlib_str_replace t s r) = smtlib_str_len t"
```

Fig. 2: Rare rule and corresponding lemma.

Figure 2 shows an example of a Rare rule (which simplifies the length of the result of a string replacement) and the Isabelle lemma generated from it by IsaRare. Roughly speaking, a rule with parameters $x_1, \ldots, x_m$, definition list $((y_1\ d_1)\ \cdots\ (y_n\ d_n))$, condition c, match expression s, and target expression t is converted by IsaRare into a lemma of the form $\forall x_1, \ldots, x_m.\ (c \Rightarrow s = t)\sigma$ where σ is the substitution $\{y_1 \mapsto d_1, \ldots, y_n \mapsto d_n\}$. Type inference in Isabelle is used to suitably instantiate the `?` wildcards in any approximate sorts in the rules.

Next we discuss the main challenges we encountered while implementing the translation from Rare to Isabelle.

4.1 Adding New Theories

Since IsaRare uses Isabelle's SMT-LIB parser, it was necessary to extend it to handle SMT theories not previously supported and, in case there was no corresponding Isabelle theory, to define new types, definitions and theorems corresponding to the SMT-LIB theory. For sets and arrays, Isabelle already provides the required data structures (`Set.set` and `Map.map` respectively) and definitions (e.g., `union`, and `store`). Translation from the SMT operators and types is thus straightforward, requiring only simple extensions to the parser.

The SMT-LIB parser also had to be extended for the operators and sorts of the SMT-LIB theory of strings. String terms are represented with Isabelle's `HOL.string`, and regular expressions are represented as sets of strings. We developed a new theory with auxiliary definitions and theorems meant to facilitate the proving of lemmas generated by IsaRare. Since strings are defined as lists of characters, we were able to reuse many list operators for our definitions. For example, string concatenation is defined as concatenation of lists.

As mentioned, bit-vectors are encoded in Isabelle using the `word` type, which represents integers modulo 2^n, where n is a type parameter (see Subsection 4.3). Isabelle has support for reasoning about this type, but we still had to provide a number of extensions. For example, to translate bit-vector rewrite rules, we had to extend Isabelle's SMT-LIB parser significantly. We added support for all of the standard SMT-LIB operators, as well as some additional operators that cvc5

```
(define-rule bv-extract-extract
 ((x ?BitVec) (i Int) (j Int) (k Int) (l Int)))
 (extract l k (extract j i x)))
 (extract (+ i l) (+ i k) x))
```

(a) A Rare rule

```
t0 = (extract j i x) ∧
size t0 = j + 1 - i ∧
t1 = (extract l k t0) ∧
size t1 = l + 1 - k ∧
t2 = (extract (i+l) (i+k) x) ∧
size t2 = (i+l) + 1 - (i+k) ∧
j < size x ∧ 0 ≤ i ∧ i ≤ j ∧
l < size t0 ∧ 0 ≤ k ∧ k ≤ l ∧
(i+l) < size x ∧ 0 ≤ (i+k) ∧
(i+k) ≤ (i+l)
```

(b) Additional Assumptions

Fig. 3: Implicit Assumption Generation

supports, such as `bvuaddo` (which checks for overflow from unsigned addition). It was also necessary to add several new definitions and basic theorems to Isabelle, for example for reasoning about the `extract` operator.

4.2 Mismatch between Isabelle and SMT-LIB operators

An important challenge for the translation concerns the mismatch between SMT-LIB operators and Isabelle functions. One of the main difficulties concerns *implicit* assumptions. As an example, consider the bit-vector extract operator. The term `(extract` i j t`)` denotes the sub-vector of bit-vector t from index i through index j, where i is the more significant index. SMT-LIB specifies that the second index j must be at most i, and both indices must be in the range $[0, n)$, where n is the bit-width of t — making the result a bit-vector of width $i + 1 - j$. These assumptions are necessary to correctly capture the semantics of SMT-LIB's `extract` since the extract operator in Isabelle is more permissive.

There are several ways to address this issue. First, we could make the implicit assumptions explicit in the Rare rules. However, this would be tedious and error-prone and would greatly clutter the Rare rules. It is also superfluous to always manually add them since the constraints are inherent in the SMT-LIB semantics. A second option is to write custom definitions for SMT-LIB operators in Isabelle that exactly match the SMT-LIB semantics (i.e., are undefined if the implicit assumptions do not hold). The main disadvantage of this approach is that it complicates proving the translated Rare rules, as those proofs cannot directly use any existing Isabelle lemmas that use the standard definitions. It also works against one of our long-term goals, which is to be able to use proof reconstruction to provide proofs for Isabelle conjectures, conjectures which will naturally use the existing Isabelle operators.

The last option, which we adopted, is to automatically add the implicit assumptions during the translation of Rare rules to Isabelle lemmas. This does make the lemmas a bit more complicated, but it is the minimal complexity needed to bridge the semantic gap between the two extract operators. And, we can be confident that these implicit assumptions will easily be discharged when using the lemmas for proof reconstruction, since SMT proofs only use operators

in ways that are consistent with SMT-LIB semantics (unless there is a bug, in which case proof reconstruction *should* fail). Figure 3 shows an example of a RARE rule with three applications of the extract operator, together with the assumptions added by IsaRARE.

In a few cases, we had to fall back on the custom definition approach. For example, we had to do this for the bit-vector `concat` operator for bit-vector concatenation. To see why, note that the SMT-LIB operator can take two or more arguments (abbreviating nested binary applications), each with arbitrary bit-width. Recall that the `:list` annotation in RARE can be used to specify a variable number of arguments. There is no way to even state lemmas corresponding to rewrite rules involving concatenations of a variable number of arguments in Isabelle using its built-in binary concatenation operator. For this case, we thus define a custom concatenation operator that matches the SMT-LIB semantics. The implicit assumption that the bit-width of the result is the sum of the bit-widths of the arguments is embedded in the custom definition. Using the new definition, we can translate the problematic rules into Isabelle lemmas. As expected, proving these lemmas requires extra work. Specifically, it requires formulating and proving bridging theorems between Isabelle's built-in concatenation operator and the new one we defined.

4.3 Supporting Approximate Sorts

With the addition of approximate sorts to RARE, we had to extend Isabelle's SMT-LIB translator to support them. We observe that Isabelle/HOL is not based on a dependently-typed logic. However, it supports an encoding of sorts depending on integer values into polymorphic types with parameters that range over types expressing ordinals. In particular, bit-vectors of width w are represented by the type `(n word)` of integers modulo 2^w; for instance, `3::(8 word)` represents an integer with value 3 modulo 2^8. In fact, thanks to polymorphism, it is possible for the bit-width to be a type variable (e.g., `3::('a::len word)`). Note that this is more precise than allowing the bit-width in the type to be completely unknown, as in approximate sorts: with type parameters one can state, for instance, that two terms of unknown bit-width have the same width, whereas two terms both of sort `?BitVec` may have different bit-widths.

Conveniently then, all the approximate sorts in RARE correspond to polymorphic types in Isabelle. For instance, `?BitVec` corresponds to `'a word` and `?Array` corresponds to `('a,'b) map` where `'a` and `'b` are type variables. During parsing, each occurrence of a approximate sort is converted into an instance of the corresponding polymorphic type obtained by instantiating each sort variable with a fresh dummy type. For some bit-vector operators, the output sort is dependent on the input sorts (e.g., `extract` and `concat` as mentioned above). For applications of such operators, we also use a dummy type for the bit-width of each argument for which the width is not known. Once translation is done, we use Isabelle's type inference algorithm to concretize each dummy type to a monomorphic one. For example, during translation of the rule `bv-ugt-eliminate` below, the variables `x` and `y` would both be assigned dummy types.

```
(define-rule bv-ugt-eliminate ((x ?BitVec) (y ?BitVec))
  (bvugt x y)   (bvult y x)
)
```

However, `bvugt` requires that both of its arguments be bit-vectors of the same width in SMT-LIB. This restriction is either already present in the definition in Isabelle that we map an operator to, or added during parsing as an implicit assumption, as we describe in Section 4.2. The type inference algorithm then computes the most general type for `x` and `y` that satisfies all assumptions. In this case, it correctly infers that they are bit-vectors of arbitrary but equal bit-width.

4.4 List Parameters

As mentioned earlier, SMT-LIB supports multi-arity syntax for certain binary operators, and Rare supports a variable number of arguments via the `:list` annotation. In contrast, in Isabelle all operators are fixed-arity. To facilitate the translation in these cases we added a new datasort, `'a rare_ListVar`, with a single constructor `ListVar :: 'a list → 'a rare_ListVar` to encapsulate multiple arguments in a list. We also introduced two second-order operators, called `rare_list_left` and `rare_list_right`, to encode Rare left-associative and right-associative operators, respectively. As an example, a Boolean term of the form `(and` $x_1 \ \cdots \ x_n \ y \ z$`)` is translated to the Isabelle term `(rare_list_right (∧) (ListVar [`$x_1, \ldots, x_n$`]) (`$y \wedge z$`))`. The `rare_list_left` and `rare_list_right` functionals fold the operator passed as first argument over the list stored in their second argument to obtain properly nested binary applications. For example, if $n = 2$, the Isabelle term above is translated to $(x_1 \wedge (x_2 \wedge (y \wedge z)))$.

For every multi-arity SMT-LIB operator, we prove that it can be built up from Isabelle's built-in binary version using `fold(r)` functions. For Rare rules with list parameters, these *transfer lemmas* become part of the correctness proof automatically generated by IsaRare. When proving the corresponding lemma, we can take advantage of the many lemmas in Isabelle's libraries about fold functions without having to know the internals of the translation process.

If we have a Rare rule in which all arguments to an operator are lists, we must handle the special case when the lists are all empty. When the operator has an identity element, we return that. For example, applications of `and` to just empty lists are translated as standing for `true`. So far, we have only encountered one operator without an identity: bit-vector concatenation. Since neither SMT-LIB nor Isabelle support bit-vectors of bit-width 0, for that operator, we explicitly add an assumption ruling out the case where all lists are empty.

4.5 Writing Lemmas and their Proofs

To generate a lemma from a Rare rewrite rule, IsaRare first introduces the parameters with their types using Isabelle's **fixes** construct. Next, it generates the statement of the lemma, the *goal*, which states that the implicit assumptions and conditions imply the equality of the match and target terms. The types of any

bit-vector constants are fully specified (via type ascription), because otherwise the lemma may be too general and not hold.

Lastly, IsaRare adds an Isabelle proof of the lemma. For lemmas that do not contain lists, this is simply a call to the main automatic tactic `auto`. Otherwise, the list constructs are eliminated as explained above, and any transfer lemmas are applied to the resulting terms. This ensures that goals will not contain any IsaRare list definitions. We then invoke induction for every list and use the `simp_all` tactic to attempt to solve and simplify the goals.

The proof is printed in *apply* style so that it can be easily modified and completed manually if Isabelle is unable to discharge all its sub-goals automatically.

4.6 Availability

IsaRare currently supports the theories of uninterpreted functions, linear arithmetic, bit-vectors, arrays, strings, and sets. It is publicly available[7] under the BSD 3-Clause license. We plan to submit IsaRare to the Archive of Formal Proofs [20]. We have also been working with the Isabelle maintainers to have our extensions to Isabelle itself (e.g., to the SMT-LIB parser) included in the official Isabelle distribution. Many features were already included in the latest release. IsaRare requires the Word_Lib library (which is also included in the Archive of Formal Proofs) if it is used on Rare rules containing bit-vector operators not present in Isabelle itself.

5 Evaluation and Experience

We used IsaRare to help develop, translate, and verify new Rare rewrite rules. These rules were designed to address coarse-grained rewrite steps appearing in cvc5 proofs, i.e., steps that could not be elaborated into fine-grained steps using the existing Rare rules and the approach mentioned in Section 2.2. In this section, we report on this experience and also discuss challenges arising from particular rewrites and theories.

5.1 Impact of New Rewrites on cvc5 Proof Holes

Previous work developed 85 Rare rules for cvc5 [28]. For our evaluation, we ran cvc5 with these plus our 163 new rules, bringing the total number of Rare rules in the cvc5 database to 248. We evaluated the impact of the new rules on cvc5's ability to produce fine-grained proof steps by comparing the *success rate* of the elaboration (i.e., percentage of rewriting proof steps that are successfully elaborated into fine-grained steps) before and after the addition of the new rules. We ran cvc5 on 70,709 unsatisfiable benchmarks, as determined by cvc5 [2, Sec. 4], in the SMT-LIB logics containing quantifier-free problems with equality and uninterpreted functions, arrays, linear arithmetic, strings, and bit-vectors.

[7] https://github.com/cvc5/IsaRARE

	rewrites			
theory	**old**	**new**	**proven**	**autoproven**
EUF	22	43	43	37
Arithmetic	23	22	22	14
Sets	0	7	7	7
Arrays	0	4	4	4
Strings	40	57	57	37
Bit-vectors	0	115	84	62

Table 1: Rule and rule verification counts per theory

The results were generated with a cluster equipped with 16 x Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz, 62.79 GiB RAM machines, with one core per solver/benchmark pair, 1200s time limit, and 8gb memory limit.

For string benchmarks (the only set evaluated in [28]), the success rate went from 92% to 98%. Results on the logics with equality and uninterpreted functions, arrays, and linear arithmetic were similar. By far the most challenging theory, in terms of rewrite rules, is the bit-vector theory. Prior to our work, there were no Rare rules for this theory, so no bit-vector rewrite steps could be turned into fine-grained steps. With our 115 new Rare rules for bit-vectors, 92% of coarse-grained bit-vector rewrite steps are successfully elaborated into fine-grained steps. We see this as tremendous progress towards full fine-grained proofs for bit-vector problems.

5.2 Translating and Verifying Rewrites

In Table 1, we list the number of new rules in each theory, distinguishing between how many were there before (old) and the total including both the old rules and our new rules (new).[8] We also show how many of the lemmas we have successfully proven and how many of these were done automatically, i.e., either by the proof suggested by IsaRare or by a single call to Sledgehammer. The **proven** column shows that all non-bit-vector rules as well as most of the bit-vector rules have now been proven. The numbers in the last column show that most of the proofs were provided automatically by IsaRare.

For the theory of strings, the number of lemmas automatically proven is not clear-cut. For other theories, libraries with useful background lemmas already existed, but for strings we had to add many new general-purpose lemmas ourselves and then decide whether these should count as background lemmas or as part of the proof effort for a rewrite rule. We were rather conservative in that decision, i.e., we did not count a lemma as automatically proved if it used a lemma whose classification as a background lemma was in doubt. Many of the

[8] Consolidation in the set of arithmetic rules actually resulted in one fewer rule than existed previously.

translated string rewrites had to be proved manually because they required induction on string length, especially since many operators are defined inductively. However, we found that most of these manual proofs were fairly easy once an appropriate induction variable was selected.

There are no performance issues—IsaRARE translates most files in milliseconds. Even for our biggest RARE database, the one containing bit-vector rules, IsaRARE took only around 1-2 seconds on our machine.

5.3 Bugs Found in String Rules

We found several bugs in the existing RARE rules for strings by using Isabelle's counterexample finder Nitpick [10] on the translated Isabelle lemmas. We diagnosed and fixed each of them, so that now they can all be verified.[9] The bugs fall into three main categories.

Misinterpreted Semantics: The `str.substr` operator takes three arguments and returns the substring of the first argument, starting at the position given by the second argument, and continuing for the number of characters specified by the third argument. The following (corrected) rule simplifies a substring expression to the empty string whenever the third argument is 0 or negative.

```
(define-cond-rule str-substr-empty-range ((x String) (n Int) (m Int))
  (>= 0 m)   (str.substr x n m)   "")
```

However, the first version of the rule had the wrong condition: `(>= n m)` rather than `(>= 0 m)`. This is likely due to the rule's author mistaking the third argument of `str.substr` for an absolute index instead of a relative offset.

Forgotten Condition: The corrected rule below says that, under some assumptions, the length of a substring term is equal to the offset (third) argument.

```
(define-cond-rule str-len-substr-in-range ((s String) (n Int) (m Int))
  (and (>= n 0) (>= m 0) (>= (str.len s) (+ n m)))
  (str.len (str.substr s n m))   m)
```

The earlier version of the rule did not include the condition `(>= m 0)`. This however, makes it unsound, because according to the semantics of str.substr, if the offset is negative, the result is just the empty string. This led to a counterexample with a negative value for `m`. Note that this condition is not automatically added by IsaRARE since `str.substr` *is* defined for negative offsets.

Misunderstanding the Rewrite: One rule was designed to closely mirror a piece of CVC5 code implementing a rewrite, but it failed to properly capture all cases. The code involved included several conditionals resulting in two different ways a term could be rewritten. The original rule only captured one of the two cases and even missed one of the conditions for the case it included. Since this rule was quite complex and was only incorrect for some corner cases, it would have been challenging to find this bug without our verification effort.

[9] Fortunately, none of the bugs in rules corresponded to buggy code in CVC5 itself. However, CVC5 could have used those rules to construct incorrect proofs.

5.4 Bit-vector Rewrite Rules

Bit-vector theory solvers make extensive use of rewriting, employing large numbers of rewrite rules. In order to define Rare rules for cvc5's bit-vector theory, we began by analyzing the cvc5 rewriting code, which implements a total of 99 rewrite methods. We then wrote Rare rules to try to capture the behavior of these methods. There are 5 methods that are too complex to be captured by Rare (or by any straightforward extension of it). For each of these, we instead added new hard-coded proof rules to the cvc5 proof rule database.[10] These hard-coded proof rules are not included in Table 1, but they *are* used to help demonstrate the overall progress on SMT-LIB proofs (Section 5.1). The long-term plan for reconstruction of proofs using these rules is to write custom Isabelle tactics for reconstructing those proof steps.

Unlike with the string rules, where we applied IsaRare to already-written rules, we used IsaRare extensively to help debug the bit-vector rules as they were being written. We were able to quickly and easily find many kinds of mistakes this way. For example, rule authors mixed up `bvneg` (unary 2's complement negation) and `bvnot` (bit-wise Boolean negation). In other cases, rules used inconsistent bit-widths. The type inference that IsaRare performs is particularly helpful in such cases, as it is stricter than the cvc5 Rare parser.

Many of the bit-vector rules can be proved automatically, but others must be proved manually and are quite challenging, especially those involving signed arithmetic or division. Despite this, as shown in Table 1, the process of manually proving the full set of bit-vector lemmas is largely complete. This is important for our long-term goal of reconstructing SMT proofs in Isabelle.

6 Conclusion

We presented IsaRare, a tool providing an automatic pipeline for verifying rewrite rules. We showed the effectiveness of our approach by proving the correctness of a large number of rewrite rules used in cvc5 proofs. Our experiments show that many lemmas can be proved with minimal user interaction.

This work is also part of a long-term project that aims to further automate proof search in Isabelle. The goal is to be able to reconstruct any cvc5 proof in Isabelle's internal inference engine. This, of course, also includes reconstructing rewrite steps. The lemmas IsaRare generates are directly applicable to this effort. We plan to provide a detailed description and evaluation of this larger effort in future work.

Data Availability Statement The datasets generated and analyzed during the current study are available in the Zenodo repository: https://zenodo.org/records/10048664 [24].

[10] This is analogous to the handling of polynomial normalization in [28].

References

1. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–9. IEEE (2018)
2. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., et al.: cvc5: a versatile and industrial-strength smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022)
3. Barbosa, H., Blanchette, J.C., Fleury, M., Fontaine, P.: Scalable fine-grained proofs for formula processing. Journal of Automated Reasoning **64**(3), 485–510 (2020)
4. Barbosa, H., Reynolds, A., Kremer, G., Lachnitt, H., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Viswanathan, A., Viteri, S., Zohar, Y., Tinelli, C., Barrett, C.: Flexible proof production in an industrial-strength SMT solver. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning. pp. 15–35. Springer International Publishing, Cham (2022)
5. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at `www.SMT-LIB.org`
6. Barrett, C., Fontaine, P., Tinelli, C.: SMT-LIB Version 3.0 - Preliminary Proposal (2021), `https://smtlib.cs.uiowa.edu/version3.shtml`
7. Barrett, C., de Moura, L., Fontaine, P.: Proofs in satisfiability modulo theories. In: Delahaye, D., Woltzenlogel Paleo, B. (eds.) All about Proofs, Proofs for All, Mathematical Logic and Foundations, vol. 55, pp. 23–44. College Publications, London, UK (Jan 2015)
8. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1267–1329. IOS Press (2021)
9. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Automated Deduction – CADE-23. pp. 116–130. Springer Berlin Heidelberg (2011)
10. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving. pp. 131–146. Springer Berlin Heidelberg (2010)
11. Böhme, S., Fox, A.C., Sewell, T., Weber, T.: Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In: International Conference on Certified Programs and Proofs. pp. 183–198. Springer (2011)
12. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving. pp. 179–194. Springer, Berlin, Heidelberg (2010)
13. Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., Fontaine, P.: veriT: an open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) Automated Deduction – CADE-22. pp. 151–156. Springer (2009)
14. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) Model Checking Software. pp. 248–254. Springer (2012)
15. Cook, B.: Formal reasoning about the security of Amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 38–47. Springer (2018)

16. Enderton, H.B.: A mathematical introduction to logic. Elsevier (2001)
17. Fleury, M.: Optimizing a verified SAT solver. In: Badger, J.M., Rozier, K.Y. (eds.) NASA Formal Methods. Lecture Notes in Computer Science, vol. 11460, pp. 148–165. Springer (2019)
18. Fleury, M., Schurr, H.J.: Reconstructing veriT proofs in isabelle/HOL. Electronic Proceedings in Theoretical Computer Science **301**, 36–50 (2019)
19. Hoenicke, J., Schindler, T.: A simple proof format for SMT. In: Déharbe, D., Hyvärinen, A.E.J. (eds.) International Workshop on Satisfiability Modulo Theories (SMT). CEUR Workshop Proceedings, vol. 3185, pp. 54–70. CEUR-WS.org (2022)
20. Jaskelioff, M., Merz, S.: Proving the correctness of disk paxos. Archive of Formal Proofs (June 2005), `https://isa-afp.org/entries/DiskPaxos.html`, Formal proof development
21. Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: Certistr: a certified string solver. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Produms and Proofs. pp. 210–224. Association for Computing Machinery (2022)
22. Katz, G., Barrett, C., Tinelli, C., Reynolds, A., Hadarean, L.: Lazy proofs for DPLL (T)-based SMT solvers. In: Piskac, R., Talupur, M. (eds.) 2016 Formal Methods in Computer-Aided Design (FMCAD). pp. 93–100. IEEE (2016)
23. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: seL4: Formal verification of an OSa kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 207–220. Association for Computing Machinery (2009)
24. Lachnitt, H., Fleury, M., Aniva, L., Reynolds, A., Barbosa, H., Noetzli, A., Barrett, C., Tinelli, C.: IsaRare: Automatic Verification of SMT Rewrites in Isabelle/HOL (Oct 2023), `https://doi.org/10.5281/zenodo.10048664`
25. de Moura, L., Bjørner, N.: Proofs and refutations, and Z3. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) Workshops. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008)
26. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
27. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer (2002)
28. Nötzli, A., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Reconstructing fine-grained proofs of rewrites using a domain-specific language. In: Griggio, A., Rungta, N. (eds.) 2022 Formal Methods in Computer-Aided Design (FMCAD). p. 65 (2022)
29. Otoni, R., Blicha, M., Eugster, P., Hyvärinen, A.E.J., Sharygina, N.: Theory-specific proof steps witnessing correctness of SMT executions. In: 2021 58th ACM/IEEE Design Automation Conference (DAC). pp. 541–546. IEEE (2021)
30. Paulson, L.C., Nipkow, T., Wenzel, M.: From LCF to Isabelle/HOL. Formal Aspects of Computing **31**(6), 675–698 (2019)
31. Schurr, H., Fleury, M., Desharnais, M.: Reliable reconstruction of fine-grained proofs in a proof assistant. In: Platzer, A., Sutcliffe, G. (eds.) Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 12699, pp. 450–467. Springer (2021)
32. Schurr, H.J., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: Towards a generic SMT proof format (extended abstract). Electronic Proceedings in Theoretical Computer Science **336**, 49–54 (2021)

33. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: ACM (ed.) Proceedings of Scheme and Functional Programming Workshop (2006)