GRAT: a formally verified (UN)SAT proof checker

Proof Checker Proposal

Peter Lammich
FMT Group, EEMCS department
University of Twente
Enschede, The Netherlands
p.lammich@utwente.nl 0000-0003-3576-0504

***Abstract*—We propose the GRAT proof checker toolchain as a verified proof checker suitable for SAT competitions. It accepts proofs in the DRAT format, and is verified down to a functional implementation in Standard ML. On benchmarks drawn from recent SAT competitions, it's performance is similar to that of drat-trim.**

I. Introduction

The GRAT toolchain accepts DRAT (ASCII and binary format) as input. The result is formally verified with Isabelle/HOL, down to the integer sequence representing the formula. The trusted code base of the verification is Isabelle/HOL's kernel and code generator, compilation and running of the extracted Standard ML code with MLton, and a thin command line wrapper and formula parser written in Standard ML.

Our tool chain follows a two step approach, with a highly optimized but unverified first step, and a formally verified second step. As the first step only acts as certificate preprocessor, it is not part of the trusted code base.

On a set of benchmarks drawn from the 2016 and 2017 SAT competitions, our full toolchain performed faster than the unverified (then state-of-the-art) tool *drat-trim*. We have confirmed that our tool is still usable for modern SAT competitions, by testing it on benchmarks from the 2022 SAT competition.

A detailed description can be found in [2], [3] and [4]. Here, we briefly summarize the main aspects, and report on the new set of benchmarks.

GRAT's webpage is https://www21.in.tum.de/~lammich/grat/, and the project is maintained as part of the IsaFOL repository https://bitbucket.org/isafol/isafol/src/master/GRAT/. Download and build instructions are on the webpage.

II. Proof Format

Our toolchain supports the de-facto standard DRAT-format as input [5].

This is then processed by the unverified *gratgen* tool, which produces a certificate enriched with unit propagation information, in the GRAT-format. The GRAT certificate and the original formula is then passed to the verified *gratchk* tool, which either confirms unsatisfiability of the formula by printing the status line *s VERIFIED UNSAT*, or yields an error.

In the following, we sketch the GRAT-format.

Each clause is identified by a unique positive ID. The clauses of the original formula implicitly get the IDs $1 \dots N$. The lemma IDs explicitly occur in the certificate.

For memory efficiency reasons, we store the certificate in two parts: The lemma file contains the lemmas, and is stored in DIMACS format. During certificate checking, this part is entirely loaded into memory. The proof file contains the hints and instructions for the certificate checker. It is not completely loaded into memory but only streamed during checking.

The proof file is a binary file, containing a sequence (stored in reverse order) of 32 bit signed integers in 2's complement little endian format. The sequence is interpreted according to the following grammar:

```
proof      ::= rat-counts item* conflict
literal    ::= int32 != 0
id         ::= int32 > 0
count      ::= int32 > 0
rat-counts ::= 6 (literal count)* 0
item       ::= uprop | del | rup-lem | rat-lem
uprop      ::= 1 id* 0
del        ::= 2 id* 0
rup-lem    ::= 3 id id* 0 id
rat-lem    ::= 4 literal id id* 0 cand-prf* 0
cand-prf   ::= id id* 0 id
conflict   ::= 5 id
```

The checker maintains a *clause map* that maps IDs to clauses, and a *partial assignment* that maps variables to true, false, or undecided. Partial assignments are extended to literals in the natural way. Initially, the clause map contains the clauses of the original formula, and the partial assignment maps all variables to undecided. Then, the checker iterates over the items of the proof, processing each item as follows:

- `rat-counts` This item contains a list of pairs of literals and the count how often they are used in RAT proofs. This map allows the checker to maintain lists of RAT candidates for the relevant literals, instead of gathering the possible RAT candidates by iterating over the whole clause database for each RAT proof, which is expensive. Literals that are not used in RAT proofs at all do not occur in the list. This item is the first item of the proof.
- `uprop` For each listed clause ID, the corresponding clause is checked to be unit, and the unit literal is assigned to true. Here, a clause is unit if the unit literal is undecided, and all other literals are assigned to false.
- `del` The specified IDs are removed from the clause map.

- `rup-lem` The item specifies the ID for the new lemma, which is the next unprocessed lemma from the lemma file, a list of unit clause IDs, and a conflict clause ID. First, the literals of the lemma are assigned to false. The lemma must not be blocked, i.e. none of its literals may be already assigned to true[1]. Note that assigning the literals of a clause C to false is equivalent to adding the conjunct $\neg C$ to the formula. Second, the unit clauses are checked and the corresponding unit literals are assigned to true. Third, it is checked that the conflict clause ID actually identifies a conflict clause, i.e. that all its literals are assigned to false. Finally, the lemma is added to the clause-map and the assignment is rolled back to the state before checking of the item started.
- `rat-lemma` The item specifies a pivot literal l, an ID for the lemma, an initial list of unit clause IDs, and a list of candidate proofs. First, as for *`rup-lemma`*, the literals of the lemma are assigned to false and the initial unit propagations are performed. Second, it is checked that the provided RAT candidates are exhaustive, and then the corresponding *`cand-prf`* items are processed: A *`cand-prf`* item consists of the ID of the candidate clause D, a list of unit clause IDs, and a conflict clause ID. To check a candidate proof, the literals of $D \setminus \{\neg l\}$ are assigned to false, the listed unit propagations are performed, and the conflict clause is checked to be actually conflict. Afterwards, the assignment is rolled back to the state before checking the candidate proof. Third, when all candidate proofs have been checked, the lemma is added to the clause map and the assignment is rolled back.
  To simplify certificate generation in backward mode, we allow candidate proofs referring to arbitrary, even invalid, clause IDs. Those proofs must be ignored by the checker.
- `conflict` This is the last item of the certificate. It specifies the ID of the conflict clause found by unit propagation after adding the last lemma of the certificate (*root conflict*). It is checked that the ID actually refers to a conflict clause.

III. Evaluation

A. Usage Example

We give a simple example on how to use our toolchain:

To verify that a formula stored in the DIMACS file `unsat.cnf` is unsatisfiable, proceed as follows:

```
# Create a (binary) drat-file
> kissat -q unsat.cnf unsat.drat
s UNSATISFIABLE
# Process into proof (gratp) and lemmas (gratl) file
> gratgen unsat.cnf unsat.drat \
    -o unsat.gratp -l unsat.gratl -b
s VERIFIED
# Check against original formula
> gratchk unsat unsat.{cnf,gratl,gratp}
s VERIFIED UNSAT
```

[1]Blocked lemmas are useless for unsat proofs, such that there is no point to include them in the certificate.

To verify that a formula stored in the DIMACS file `sat.cnf` is satisfiable, proceed as follows:

```
# Produce variable assignment,
# as 0-terminated list of literals
> kissat -q sat.cnf | grep "^v" \
   | sed -re 's/^v//g' > sat.vars
# Check against original formula
> gratchk sat sat.{cnf,vars}
s VERIFIED SAT
```

B. MLtons Memory Manager

When running gratchk on machines with a lot of memory, we ran into two problems with MLtons default memory manager: First it will take half of the machine's memory before even starting to garbage collect. And, second, when it garbage collects, it will try to keep allocated 8 times the live memory size. Both behaviours are problematic: small problems will consume huge amounts of memory, making it impossible to verify many small problems in parallel on the same machine. Also, most of the memory that gratchk consumes is the storage for the formula and lemmas. Once the checking starts, only little additional memory is needed. However, MLtons memory manager will try to allocate 8 times the live size, which includes the (potentially large) formula and lemmas. In practice, this led to gratchk processes being killed by the out-of-memory killer.

While there is no ideal solution currently supported by MLton, we decided to apply a simple heuristic and limit the memory available to gratchk to 10 times the formula and lemma file size, and a minimum of 1GiB. In practice, this can be achieved by system tools, or by a runtime option to MLton, e.g.:

```
> gratchk @MLton max-heap 2G -- \
    unsat unsat.{cnf,gratl,gratp}
```

C. Theoretical Complexity

Our toolchain has polynomial complexity in the size of the input (drat) certificate and formula. While we have not estimated the precise complexities, we give a rough argument that the complexity is polynomial.

The first phase, *gratgen*, iterates over each clause in the certificate, and puts it into a two-watched-literals (twl) data structure. This clearly takes polynomial time. It then iterates backwards over the clauses. For each clause, the (inverted) literals are added as units, and then unit-propagation is performed. This also takes polynomial time. In case of a RAT clause, further clauses are gathered from the available clauses, and for each of those, another unit propagation is done (again, polynomial unit propagation for linearly many clauses). After checking each clause, the twl data structure is reverted to the state before that clause (which also takes polynomial time).

The second phase, *gratchk*, repeats the actions from the first phase, but iterating in a forwards fashion, and using extra information for unit propagation. Thus, it is also clearly polynomial.

D. Empirical Evaluation

We have extensively benchmarked our toolchain in [4], where we also compared it against the then-current versions of *drat-trim* and LRAT [1].

Our tool has not significantly changed since then, and we refer the reader to [4] for those results.

To check if our tool is still usable, we have run it on problems from the 2022 SAT competition's main track. We considered the winning solver Kissat_MABHyWalk, and the highest ranked non-Kissat based solver SeqFROST-ERE-All. We ran the solvers on all unsatisfiable problems they could solve in the competition to regenerate the certificates, and then used GRAT to verify the results. We benchmarked two configurations for gratgen: single-threaded and 8 parallel threads. Previous experiments have shown that more than 8 threads do not bring significant speedup.

1) Verified Problems: First of all, we could verify all 146 problems for Kissat and all 138 problems for SeqFROST. The single-threaded gratgen timed out on one Kissat problem, though.

2) Solving vs. Verification time: We compare the solving time with the verification time. Let t_s be the solving time, and t_v be the verification time, we compute, for each problem the ratio $r = t_v/t_s$, and then count for what percentage of the problems this ratio is less than .5, 1, 2, and 4. This is a sensible measure, as we expect the verification time to be related to the difficulty of the problem, and thus the solving time. Also, it estimates the extra time required to get a verified result. The result is displayed in the following table:

	$< .5$	< 1	< 2	< 4	#problems
Kissat-j8	70.5	85.6	93.8	97.3	146
SeqFROST-j8	76.8	89.1	96.4	99.3	138
Kissat-j1	26.9	60.0	87.6	93.8	145
SeqFROST-j1	24.6	50.7	81.9	97.1	138

That is, with 8 threads, we can verify more than 80% of the problems when allowing the same time for verification as for solving. In single-threaded mode, it's still more than half of the problems. And more than 90% of the problems will be solved when allowing a factor of 2 (8 threads) or 4 (1 thread), respectively.

3) Drat Certificate vs Grat Certificate Size: Next, we compare the size of the drat certificate produced by the SAT solver to the size of the enriched (grat) certificate produced by the first phase of our tool. This is of concern as the certificates have to be stored on disk, and thus, should not be excessively big. As for the time, we determine the ratio grat-size over drat-size, and count the percentage of problems below certain ratios.

	$< .5$	< 1	< 2	< 4	#problems
Kissat	46.6	54.1	84.9	97.9	146
SeqFROST	50.0	54.3	81.9	97.8	138

We observe that the generated grat certificate is smaller than the original drat certificate in more than half of the cases, and rarely exceeds factor 4. This is due to the trimming heuristics in gratgen, which, similar to drat-trim, tries hard to eliminate as many useless lemmas as possible. In many cases, this elimination removes more than the extra unit-propagation information that is added by gratgen.

IV. Formal Verification

The crucial part of our toolchain is the *gratchk* tool, which takes as input the original formula and a certificate in GRAT format, and then verifies that the formula is actually unsatisfiable. It also supports a mode for verifying satisfiable formulas, which takes a list of true literals as proof.

The *gratchk* tool is written in Standard ML, and compiled using the MLton compiler. The top-level is an unverified command line interface, which interprets the commands, and parses the specified files into an array of integers. The array contains a representation of the formula, followed by a representation of the proof. It then calls the core functions `verify_sat_impl` and `verify_unsat_split_impl`, which are exported from an Isabelle formalization using Isabelle's code generator.

```
val verify_sat_impl
  : int array → nat → unit → (_, _) sum
val verify_unsat_split_impl
  : int array → ('a → int * 'a)
    → nat → nat → nat * 'a → unit → (_, _) sum
```

For these functions, we have proved the following lemmas in Isabelle:

```
theorem verify_sat_impl_correct:
  <DBi ↦_a DB>
    verify_sat_impl DBi F_end
  <λresult. DBi ↦_a DB
    * ↑(¬isl result ⟹ verify_sat_spec DB F_end)>

theorem verify_unsat_impl_correct:
  <DBi ↦_a DB>
    verify_unsat_split_impl DBi prfn F_end it prf
  <λresult. DBi ↦_a DB
    * ↑(¬isl result ⟹ verify_unsat_spec DB F_end)>
```

The preconditions of these Hoare triples state that the argument `DBi` points to an array holding the elements `DB`. This array is not changed by the functions (it occurs unchanged in the postcondition), and these Hoare-triples imply termination of the program, as well as that it does not change any memory apart from what it allocates itself.

The original formula is stored in `DB[1..<F_end]`. (`DB[0]` is used as a guard by our implementation). The result of the functions are from an exception monad, represented by a sum type. The second parts of the postconditions state that, if no exception is raised, the formula stored at `DB[1..<F_end]` is satisfiable or unsatisfiable respectively. In case of the unsat proof, the other parameters `prfn, it, prf` are used to represent the proof, but they have no influence on the statement of this lemma: regardless of their values, an accepted formula is always unsat (If we pass nonsense, however, we will likely get an exception).

To express when a formula is (un)sat, we have two (proved equivalent) specifications. The first version relies on a function *F_α* that maps lists of integers to our internal representation of SAT formulas, and the predicate *sat* that specifies if a formula is satisfiable:

```
definition verify_sat_spec DB F_end ≡
    1 ≤ F_end ∧ F_end ≤ length DB
  ∧ (let lst = tl (take F_end DB) in
       F_invar lst ∧ sat (F_α lst))

definition verify_unsat_spec DB F_end ≡
    1 ≤ F_end ∧ F_end ≤ length DB
  ∧ (let lst = tl (take F_end DB) in
       F_invar lst ∧ ¬sat (F_α lst))
```

These specifications state that *F_end* is in range, and that *DB*[*1..<F_end*] (in Isabelle: *tl (take F_end DB)*) is a valid (*F_invar*) representation of a satisfiable or unsatisfiable, respectively, formula.

To increase the trust in these specifications, we prove them equivalent to a version that only relies on basic list operations: First, we use the function *tokenize :: int list ⇒ int list list*, which splits a list into its zero-terminated components. To sanity-check this function, we prove that, for a list that ends with a zero (i.e., contains no open clause at the end), its result is the unique inverse of concatenation:

```
definition concat0 ll = concat (map (λl . l@[0]) ll)
lemma unique_tokenization:
  assumes l≠[] ⟹ last l = 0
  shows ∃₁ls. (0∉⋃set (map set ls) ∧ concat0 ls = l)
    and tokenize l = (THE ls.
          0∉⋃set (map set ls) ∧ concat0 ls = l)
```

where *THE* is the definite description operator.

Next, we define an assignment from integers to Booleans to be consistent iff a negative value is mapped to the opposite of its absolute value:

```
definition assn_consistent :: (int ⇒ bool) ⇒ bool
  where assn_consistent σ
    = (∀x. x≠0 ⟹ ¬ σ (-x) = σ x)
```

Finally, we characterize an (un)satisfiable input by the (non)existence of a consistent assignment that assigns at least one literal of each clause to true. Thus, we prove the following alternative characterizations of our specifications:

```
lemma verify_sat_spec DB F_end = (
  1≤F_end ∧ F_end ≤ length DB ∧ (
  let lst = tl (take F_end DB) in
    (lst≠[] ⟹ last lst = 0)
  ∧ (∃σ. assn_consistent σ
        ∧ (∀C∈set (tokenize 0 lst). ∃l∈set C. σ l))))

lemma verify_unsat_spec DB F_end = (
  1 < F_end ∧ F_end ≤ length DB ∧ (
  let lst = tl (take F_end DB) in
      last lst = 0
  ∧ (∄σ. assn_consistent σ
        ∧ (∀C∈set (tokenize 0 lst). ∃l∈set C. σ l))))
```

In the case of unsatisfiability, the bounds have been adjusted to exclude the empty formula, which is trivially satisfiable.

A. Trusted Code Base

Our approach relies on the correctness of the following components

- Isabelle/HOL's inference kernel.
- Isabelle/HOL's code generator to Standard ML.
- The Imperative/HOL extension of the code generator.
- The correct formalization of what a Hoare-triple means.
- The correct specification of the correctness properties.
- The command line interface and DIMACs file parser.
- The correctness of the ML compiler and execution environment.

Where possible, we have tried to keep these trusted components as simple as possible. For example, we have proved two equivalent forms of the correctness specification, and limited the unverified parser to parse the DIMACs file into an array of integers. The interpretation of these integers as list of clauses is done inside Isabelle.

References

[1] M. Heule, W. Hunt, M. Kaufmann, and N. Wetzler. Efficient, verified checking of propositional proofs. In *Proc. of ITP*. Springer, 2017.

[2] P. Lammich. Efficient verified (UN)SAT certificate checking. In *Proc. of CADE*. Springer, 2017.

[3] P. Lammich. The GRAT tool chain - efficient (UN)SAT certificate checking with formal correctness guarantees. In *SAT*, pages 457–463, 2017.

[4] P. Lammich. Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.*, 64(3):513–532, 2020.

[5] N. Wetzler, M. J. H. Heule, and W. A. Hunt. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In *Proc. of SAT 2014*, pages 422–429. Springer, 2014.