VeriPB and CakePB in the SAT Competition 2023

Bart Bogaerts Ciaran McCreesh Magnus O. Myreen Jakob Nordström Andy Oertel Yong Kiam Tan

I. Summary

The pseudo-Boolean proof format used for the proof checker VeriPB [1] supports proof logging for decision, enumeration, and optimization problems, as well as problem reformulations, all in a unified format. So far, VeriPB has been used for proof logging of enhanced SAT solving techniques [2], [3], pseudo-Boolean CDCL-based solving [4], constraint programming [5], [6], subgraph solving [7], [8], and MaxSAT solving [9], [10], and this list of applications is expected to keep growing. This description briefly summarizes how a restricted version of the proof format can be used to certify unsatisfiability of CNF formulas in the SAT competition 2023. A complete documentation of the proof format can be found at https://gitlab.com/MIAOresearch/software/VeriPB/-/blob/satcomp2023_checker/satcomp23/documentation_SAT_competition_2023.pdf.

II. Quickstart Guide for Boolean Satisfiability (SAT) Proof Logging

This section contains the bare minimum of information needed to use VeriPB and CakePB as proof checkers for Boolean satisfiability (SAT) solvers with pseudo-Boolean proof logging. A good way to learn more (in addition to reading this document) might be to study the example files in the directory `tests/integration_tests/correct/` in the repository [1] and run VeriPB with the options `--trace --useColor`, which will output detailed information about the proofs and the proof checking.

A. Running the Proof Checkers

If a SAT solver with pseudo-Boolean proof logging has solved the instance `input.cnf`, the generated proof `input.pbp` can be checked by VeriPB and CakePB by runnning the following commands:

```
# Translate to kernel format proof
veripb --cnf --proofOutput translated.pbp \
  input.cnf input.pbp

# Check the kernel proof
cake_pb_cnf input.cnf translated.pbp
```

The first command recompiles the pseudo-Boolean proof `input.pbp` into a more restricted "kernel-format" proof `translated.pbp` using VeriPB, after which the kernel proof is checked using CakePB. In case of successful recompilation, VeriPB will output:

```
# Running veripb as shown above
...
Verification succeeded
```

Upon successful proof checking, CakePB will report success on the standard output stream:

```
# Running cake_pb_cnf as shown above
s VERIFIED UNSAT
```

All errors are reported on standard error.

B. Proof Format

The syntactic format of a pseudo-Boolean proof of unsatisfiability for a CNF formula as expected by the version of VeriPB proposed for the SAT competition 2023 is

`pseudo-Boolean proof version 2.0`
`f` $\langle N \rangle$
Derivation section
`output NONE`
`conclusion UNSAT :` $\langle id \rangle$
`end pseudo-Boolean proof`

where $\langle N \rangle$ should be the number of clauses in the formula and *Derivation section* should contain the actual proof which derives contradiction as the pseudo-Boolean constraint with constraint ID $\langle id \rangle$.

In pseudo-Boolean format, a disjunctive clause like

$$x_1 \lor \overline{x}_2 \lor x_3 \tag{1a}$$

is represented as the inequality

$$x_1 + \overline{x}_2 + x_3 \geq 1 \tag{1b}$$

claiming that at least one of the literals in the clause is true (i.e., takes value 1), and this inequality is written as

```
+1 x1 +1 ~x2 +1 x3 >= 1 ;
```

in the OPB format [11] used by VeriPB. The proof checker can also read CNF formulas in the DIMACS and WDIMACS formats used for SAT solving and MaxSAT solving, respectively. For such files, VeriPB will parse a clause

```
1 -2 3 0
```

to be identical to (1b), and the variables should be referred to in the pseudo-Boolean proof file as `x1`, `x2`, `x3`, et cetera.

DRAT proofs can be transformed into valid VeriPB proofs by simple syntactic manipulations. Most of the proof resulting from a CDCL SAT solver is the ordered sequence of clauses learned during conflict analysis. Since all such clauses are guaranteed to be *reverse unit propagation (RUP)* clauses, the easiest way to provide pseudo-Boolean proof logging for a learned clause (1a) would be to write

```
rup +1 x1 +1 ~x2 +1 x3 >= 1 ;
```

in the derivation section of the pseudo-Boolean proof.

If instead the clause (1a) is a *resolution asymmetric tautology (RAT)* clause that is RAT on the literal x_1, then this is written as

```
red +1 x1 +1 ~x2 +1 x3 >= 1 ; x1 -> 1
```

in the pseudo-Boolean proof using the more general *redundance-based strengthening* rule. And if the RAT literal would instead have been $\overline{x}_2$, this would have been indicated by ending the proof line above by `x2 -> 0` instead.

Finally, in order to delete the clause (1a), the deletion command

```
del spec +1 x1 +1 ~x2 +1 x3 >= 1 ;
```

is issued. An important difference from DRAT proofs is that deletion is made also for unit clauses, i.e., clauses containing only a single literal—DRAT proof checkers typically ignore such deletion commands. Another crucial difference is that all clauses learned during CDCL execution need to be written down in the proof log, including unit clauses. If unit clauses are missing in a DRAT proof, the proof checkers will typically be helpful and silently infer and add the missing clauses. No such patching of formally incorrect proofs is offered by VeriPB.

It should be noted, though, that if all the reasoning performed by some particular SAT solver can efficiently be captured by standard DRAT proof logging, then there is no real reason to use pseudo-Boolean proof logging for that solver. Pseudo-Boolean proof logging becomes relevant only if the solver uses more advanced techniques such as, for instance, cardinality reasoning, Gaussian elimination, or symmetry breaking. We refer the reader to [2] and [3], respectively, for detailed descriptions of how to do efficient pseudo-Boolean proof logging for the latter two techniques. A detailed description of the cutting planes proof system and proof steps supported in the augmented for VeriPB and the kernel format for CakePB is available at https://gitlab.com/MIAOresearch/software/VeriPB/-/blob/satcomp2023_checker/satcomp23/documentation_SAT_competition_2023.pdf.

III. Formally Verified Proof Checking

The kernel proof checker CakePB has been formally verified in the HOL4 theorem prover [12] using the CakeML suite of tools for program verification, extraction, and compilation [13]–[15]. In this section, we present the verification guarantees for CakePB_cnf, a version of CakePB equipped with a DIMACS CNF parser frontend for UNSAT proof checking with pseudo-Boolean proof logging.

A. *Verified Correctness Theorem for* CakePB_cnf

The end-to-end verified correctness theorem for CakePB_cnf is shown in Figure 1. This theorem can be intuitively understood in four parts, corresponding to the indicated lines (2)–(5):

- The theorem assumes (2) that the CakeML-compiled machine code for CakePB_cnf is executed in an x64 machine environment set up correctly for CakeML. The definition of cake_pb_cnf_run is shown below, where the first line (wfcl cl ∧ wfFS fs ∧ ...) says the command line cl and filesystem fs match the assumptions of CakeML's FFI model. The second line says that the compiled code (cake_pb_cnf_code) is correctly set up for execution on an x64 machine.

  $$\text{cake\_pb\_cnf\_run } cl\ fs\ mc\ ms \stackrel{\text{def}}{=} \text{wfcl } cl \land \text{wfFS } fs \land \text{STD\_streams } fs \land \text{hasFreeFD } fs \land \text{installed\_x64 cake\_pb\_cnf\_code } mc\ ms$$

- Under these assumptions, the CakePB_cnf program is guaranteed to never crash (3). However, it may run out of resources such as heap or stack memory (extend_with_resource_limit ...). In these cases, CakePB_cnf will fail gracefully and report out-of-heap or out-of-stack on standard error.
- Upon termination, the CakePB_cnf program will output some (possibly empty) strings out and err to the standard output and standard error streams, respectively (4).
- The key verification guarantee (5) is that, whenever the string "s VERIFIED UNSAT" is printed to standard output, the input CNF file (first command line argument) parses in DIMACS format to a CNF which is unsatisfiable. No other output is possible on standard output; error strings are always printed to standard error.

Internally, CakePB_cnf transforms input CNF clauses (in DIMACS format) to normalized pseudo-Boolean constraints, as exemplified by (1a) and (1b). This transformation is formally verified to preserve satisfiability as part of the end-to-end correctness theorem shown in Figure 1. Note that the CakePB_cnf tool has an essentially identical correctness theorem to an existing verified Boolean unsatisfiability proof checking tool [16]. In fact, these tools share exactly the same definitions of DIMACS CNF parsing, Boolean satisfiability semantics, and all of the CakeML's standard assumptions.

B. *Complexity and Empirical Evaluation*

All of the commands in the kernel format are designed to minimize the need to search over the entire constraint database. For example, each implicational and deletion proof step can be performed in linear time with respect to the size of that step.

The only proof steps that scale linearly with respect to the size of the constraint database are redundancy and dominance-based strengthening steps. For either of these steps, the proof checker potentially needs to loop over the entire constraint database to check all the necessary proof goals. However, the maximum size of the database is linear in the size of the input formula and the proof. Therefore, the overall complexity of the verified proof checker is polynomial in the size of the input formula and proof, as required.

Table I shows an empirical evaluation of the verified proof checking pipeline on a selected suite of example proofs, generated using BreakID [17][1] and Kissat [18][2] to solve

[1]https://bitbucket.org/krr/breakid/src/veriPB/

[2]https://gitlab.com/MIAOresearch/tools-and-utilities/kissat_fork

$$\vdash \textsf{cake\_pb\_cnf\_run}\ cl\ fs\ mc\ ms \Rightarrow \tag{2}$$

$$\left.\begin{array}{l}
\textsf{machine\_sem}\ mc\ (\textsf{basis\_ffi}\ cl\ fs)\ ms \subseteq \\
\quad \textsf{extend\_with\_resource\_limit} \\
\qquad \{\ \textsf{Terminate Success}\ (\textsf{cake\_pb\_cnf\_io\_events}\ cl\ fs)\ \}\ \wedge
\end{array}\right\} \tag{3}$$

$$\left.\begin{array}{l}
\exists\, out\ err. \\
\quad \textsf{extract\_fs}\ fs\ (\textsf{cake\_pb\_cnf\_io\_events}\ cl\ fs) = \\
\qquad \textsf{SOME}\ (\textsf{add\_stdout}\ (\textsf{add\_stderr}\ fs\ err)\ out)\ \wedge
\end{array}\right\} \tag{4}$$

$$\left.\begin{array}{l}
\textsf{if}\ out = \text{«s VERIFIED UNSAT\n»}\ \textsf{then} \\
\quad \textsf{LENGTH}\ cl = 3 \wedge \textsf{inFS\_fname}\ fs\ (\textsf{EL}\ 1\ cl)\ \wedge \\
\quad \exists\, fml. \\
\qquad \textsf{parse\_dimacs}\ (\textsf{all\_lines}\ fs\ (\textsf{EL}\ 1\ cl)) = \textsf{SOME}\ fml\ \wedge \\
\qquad \textsf{unsatisfiable}\ (\textsf{interp}\ fml) \\
\textsf{else}\ out = \text{«»}
\end{array}\right\} \tag{5}$$

Fig. 1: The end-to-end correctness theorem for the CAKEML pseudo-Boolean proof checker with a CNF parser

TABLE I: Example timings for verified proof checking using VERIPB and CAKEPB_CNF. All times are in seconds.

Benchmark	VeriPB Time (s)	CakePB Time (s)
queen14_14.col.14.cnf	6.5	52.3
harder-php{...}.cnf	9.3	30.5
Pb-chnl15-16_c18.cnf	13	43.2
tseitin_n104_d3.cnf	4.2	3.9
rphp_p6_r28.cnf	123	68.2

SAT competition instances of the last years and theoretical instances.

ACKNOWLEDGEMENTS

We wish to acknowledge the monumental contributions of Stephan Gocht [19], without whom there would not have been any pseudo-Boolean proof checker.

REFERENCES

[1] "VeriPB: Verifier for pseudo-Boolean proofs," https://gitlab.com/MIAOresearch/software/VeriPB.

[2] S. Gocht and J. Nordström, "Certifying parity reasoning efficiently using pseudo-Boolean proofs," in *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, Feb. 2021, pp. 3768–3777.

[3] B. Bogaerts, S. Gocht, C. McCreesh, and J. Nordström, "Certified symmetry and dominance breaking for combinatorial optimisation," in *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI '22)*, Feb. 2022, pp. 3698–3707.

[4] S. Gocht, R. Martins, J. Nordström, and A. Oertel, "Certified CNF translations for pseudo-Boolean solving," in *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 236, Aug. 2022, pp. 16:1–16:25.

[5] J. Elffers, S. Gocht, C. McCreesh, and J. Nordström, "Justifying all differences using pseudo-Boolean reasoning," in *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, Feb. 2020, pp. 1486–1494.

[6] S. Gocht, C. McCreesh, and J. Nordström, "An auditable constraint programming solver," in *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 235, Aug. 2022, pp. 25:1–25:18.

[7] ——, "Subgraph isomorphism meets cutting planes: Solving with certified solutions," in *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, Jul. 2020, pp. 1134–1140.

[8] S. Gocht, R. McBride, C. McCreesh, J. Nordström, P. Prosser, and J. Trimble, "Certifying solvers for clique and maximum common (connected) subgraph problems," in *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, ser. Lecture Notes in Computer Science, vol. 12333. Springer, Sep. 2020, pp. 338–357.

[9] D. Vandesande, W. De Wulf, and B. Bogaerts, "QMaxSATpb: A certified MaxSAT solver," in *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR '22)*, ser. Lecture Notes in Computer Science, vol. 13416. Springer, Sep. 2022, pp. 429–442.

[10] J. Berg, B. Bogaerts, J. Nordström, A. Oertel, and D. Vandesande, "Certified core-guided MaxSAT solving," in *Proceedings of CADE-29*, 2023, accepted for publication.

[11] O. Roussel and V. M. Manquinho, "Input/output format and solver requirements for the competitions of pseudo-Boolean solvers," Jan. 2016, revision 2324. Available at http://www.cril.univ-artois.fr/PB16/format.pdf.

[12] K. Slind and M. Norrish, "A brief overview of HOL4," in *TPHOLs*, ser. LNCS, O. A. Mohamed, C. A. Muñoz, and S. Tahar, Eds., vol. 5170. Springer, 2008, pp. 28–32.

[13] Y. K. Tan, M. O. Myreen, R. Kumar, A. C. J. Fox, S. Owens, and M. Norrish, "The verified CakeML compiler backend," *J. Funct. Program.*, vol. 29, p. e2, 2019.

[14] A. Guéneau, M. O. Myreen, R. Kumar, and M. Norrish, "Verified characteristic formulae for CakeML," in *ESOP*, ser. LNCS, H. Yang, Ed., vol. 10201. Springer, 2017, pp. 584–610.

[15] M. O. Myreen and S. Owens, "Proof-producing translation of higher-order logic into pure and stateful ML," *J. Funct. Program.*, vol. 24, no. 2-3, pp. 284–315, 2014.

[16] Y. K. Tan, M. J. H. Heule, and M. O. Myreen, "cake_lpr: Verified propagation redundancy checking in CakeML," in *TACAS*, ser. LNCS, J. F. Groote and K. G. Larsen, Eds., vol. 12652. Springer, 2021, pp. 223–241.

[17] "Breakid," https://bitbucket.org/krr/breakid.

[18] "Kissat SAT solver," http://fmv.jku.at/kissat/.

[19] S. Gocht, "Certifying correctness for combinatorial algorithms by using pseudo-Boolean reasoning," Ph.D. dissertation, Lund University, Lund, Sweden, Jun. 2022.