From DRUP to PAC and Back

Daniela Kaufmann Armin Biere Manuel Kauers
Johannes Kepler University Linz, Altenbergerstr. 69, 4040 Linz, Austria
daniela.kaufmann@jku.at armin.biere@jku.at manuel.kauers@jku.at

***Abstract*—Currently the most efficient automatic approach to verify gate-level multipliers combines SAT solving and computer algebra. In order to increase confidence in the verification, proof certificates are generated. However, due to different solving techniques, these certificates require two different proof formats, namely DRUP and PAC. A combined proof has so far been missing. Correctness of this approach can thus only be trusted up to the correctness of compositional reasoning. In this paper we show how to generate a single proof in one proof format, which then allows to certify correctness using one simple proof checker. We further investigate empirically the effect on proof generation and checking time as well as on proof size. It turns out that PAC proofs are much more compact and faster to check.**

I. Introduction

Fully automated verification of gate-level multiplier circuits is still considered to be hard. The currently most effective approach relies on computer algebra [4], [13], [14]. Whereas the authors of [4], [14] employ only algebraic reasoning, we further combine *Boolean satisfiability* (SAT) solving [13]. We conjectured in [13], that certain final stage adders are a real challenge for the computer algebra approach. On the other hand these adders can easily be verified using SAT solvers. In our approach we replacing complex adders by simple ripple carry adders (RCA). The correctness of the substitution is proved by SAT solvers and the rewritten multiplier is verified using the computer algebra approach.

We increase the trust in the verification result by generating certificates in [13], which can be checked by independent proof checkers. Since our technique relies on two different reasoning techniques, also two proof certificates in different proof formats are produced. The polynomial reduction algorithm produces an algebraic proof in the *practical algebraic calculus* (PAC) [16] and SAT solvers produce clausal proofs in the *delete reverse unit propagation* (DRUP) proof format [10]. These proofs are checked by two different proof checkers, leaving a hole in the certification argument. Compositional reasoning using interactive theorem proving [12] could close this gap but is not fully automatic.

In this work we present how these two proof formats used in [13] can be merged into one common proof format. Although this paper is tailored to the use case of [13], the proposed methods are not limited to this particular application.

We are able to convert a DRUP proof into a PAC proof. On the other hand our results for converting a PAC proof into a DRUP proof can be considered to provide a lower bound on the proof size. In the conversion we use a *satisfiability modulo theories* (SMT) encoding and thus are not able to track any rewriting employed by SMT solvers as a DRUP proof.

Our experiments generate proofs in a single proof format. It turns out that PAC proofs are superior to DRUP proofs, as DRUP proofs are around three orders of magnitude larger than PAC proofs. Additionally, as already mentioned, our DRUP proofs do not yet cover all necessary proof steps.

Supported by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), P31571-N32, SFB F5004, LIT AI Lab funded by the state of Upper Austria.

II. Preliminaries

We recapitulate the two proof formats DRUP and PAC and further summarize the state-of-the-art [13] for automatic verification of unsynthesized multiplier circuits.

A. Algebra and the PAC format

In this section we introduce basic concepts of algebra [5] and describe the PAC proof format [16].

A nonempty subset of polynomials $I \subseteq \mathbb{Z}[X]$ is called an *ideal* if $\forall\, p, q \in I : p+q \in I$ and $\forall\, p \in \mathbb{Z}[X]\ \forall\, q \in I : pq \in I$. A set $P = \{p_1, \ldots, p_s\} \subseteq \mathbb{Z}[X]$ is called a *basis* of I if $I = \{p_1 q_1 + \cdots + p_s q_s \mid q_1, \ldots, q_s \in \mathbb{Z}[X]\}$. We then say I is generated by P and write $I = \langle P \rangle$.

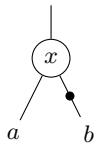
Let $f \in \mathbb{Z}[X]$ and $P \subseteq \mathbb{Z}[X]$. We are interested whether the polynomial equation $f = 0$ is implied by the equations $p = 0$ with $p \in P$. This question is also called ideal membership problem: Given f and P as above decide whether $f \in \langle P \rangle$.

We focus on gate-level circuit verification, where all variables $x \in X$ represent logic gates and thus take only values in $\{0, 1\}$. This is enforced by *Boolean value constraints* of the form $x(1-x) = 0$. Let $B(X) = \{x(1-x) \mid x \in X\} \subseteq \mathbb{Z}[X]$ be the set of Boolean value constraints for X. Each gate of the circuit is encoded by a polynomial relation, called *gate polynomial*, which are collected in P. Consequently the ideal membership problem we actually want to solve is formulated as: Given $f \in \mathbb{Z}[X]$ and $P \subseteq \mathbb{Z}[X]$, decide whether $f \in \langle P \cup B(X) \rangle$.

The practical algebraic calculus (PAC) format allows to capture the derivation of an equation $f = 0$ from a given set of polynomials equations P and thus $f \in \langle P \cup B(X) \rangle$.

Proofs are sequences of proof rules, which model the ideal properties, where each rule has the following form:

$+ : p_i, p_j, p_i + p_j;$	p_i, p_j appearing earlier in the proof or are contained in P and $p_i + p_j$ being reduced by $B(X)$
$* : p_i, q, qp_i;$	p_i appearing earlier in proof or in P and $q \in \mathbb{Z}[X]$ being arbitrary and qp_i being reduced by $B(X)$



	Propostional Formula	Polynomial Relation
	$(x \leftrightarrow a \wedge \overline{b}) = \top$	$-x + a(1-b) = 0$
CNF	$(x \vee \overline{a} \vee b) = \top$	$(1-x)a(1-b) = 0$
	$(\overline{x} \vee a) = \top$	$x(1-a) = 0$
	$(\overline{x} \vee \overline{b}) = \top$	$xb = 0$

Fig. 1. Different encodings of the AIG node $x = a \wedge \overline{b}$

B. SAT and the DRUP format

We briefly introduce the SAT problem and its common proof formats, following [10]. The SAT problem seeks for an assignment such that a formula F evaluates to **true**.

A clause C is *redundant* w.r.t. a formula F, if $F \wedge C$ is satisfiable iff F is satisfiable. Redundant clauses are for example derived using *resolution*: Given two clauses $C_1 = (a \vee x)$ and $C_2 = (\overline{a} \vee y)$, the clause $C = (x \vee y)$ can be resolved.

A further technique used in SAT solvers is called *unit propagation*: If a formula F contains a unit clause $C = l$, remove all clauses containing l and all occurrences of $\overline{l}$.

If a formula is satisfiable a satisfying assignment can act as witness. However if the formula is unsatisfiable more involved reasoning is required to derive proofs of unsatisfiability, also called *refutation*. Standard refutation proof formats are either resolution proofs or clausal proofs. Clausal proofs are easier to generate and are more compact than resolution proofs.

The most basic clausal proof format is *reverse unit propagation* (RUP) [7]. We say C is a RUP *clause* if $F \wedge \overline{C}$, with $\overline{C}$ being the negation of C, is unsatisfiable. RUP proofs are sequences of RUP clauses containing the empty clause. A *delete reverse unit propagation* (DRUP) [9] proof extends RUP by adding deletion information [18]. DRUP can further be extended to *deletion resolution asymmetric tautology* (DRAT) [8] by for instance allowing to introduce new variables. Clausal DRUP proofs are checked through unit propagation. As a side effect a resolution proof [10] can be produced. The TRACECHECK format is a common resolution proof format.

C. State-of-the-art Circuit Verification

Multipliers are usually made up of three stages: (i) generation of partial products (PPG), (ii) accumulation of the partial products (PPA) and (iii) a final stage adder (FSA).

According to the state-of-the-art [13] the first two stages are easy for computer algebra, but some final stage adders, more precisely generate-and-propagate adders, are challenging for the computer algebra approach. However these adders are very easy for SAT solvers. In our technique of [13] we are given multipliers as And-Inverter-Graphs (AIG). We identify whether the final stage adder is a generate-and-propagate adder and if necessary substitute it with a simple RCA.

To verify that the original FSA is substituted with an equivalent RCA, a bit-level miter in CNF is generated, and checked by a SAT solver, which also produces a DRUP proof for certification. Correctness of the rewritten circuit is shown using computer algebra. For details see [4], [13], [14]. A PAC proof is computed alongside the polynomial reduction. In the toolflow of [13] the PAC proof is split into the ".polys" and ".pac" files, where "".polys" contains the initial set of polynomials P and ".pac" contains the PAC proof rules.

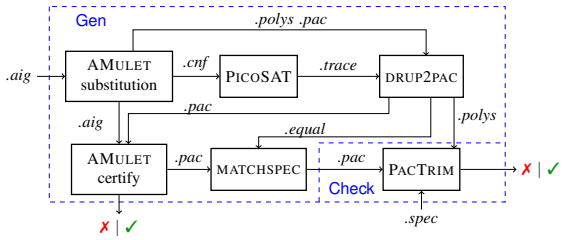


Fig. 2. Converting a DRUP proof into a PAC proof

III. FROM DRUP TO PAC

The necessary steps to merge the DRUP and the PAC proof of [13] into one single PAC proof are shown Fig. 2. Converting the DRUP proof into a PAC proof needs algebraic reasoning over the CNF encoding derived during adder substitution. As only the gate polynomials are contained in the constraints set we need to deduce the CNF encoding in PAC. Figure 1 shows an AIG node and the corresponding encodings as propositional formulas and polynomial equations. Since in a satisfiable CNF every clause needs to evaluate to **true**, the CNF can be split into a system of "clausal equations" (on the right) encoding this property. We derive the corresponding system of polynomial equations from the initial polynomial relation by simple polynomial operations. We added to the original tool AMULET of [13] the ability to derive such polynomial encodings of CNFs during adder substitution.

The generated CNF miter of the adder substitution is given to the SAT solver PICOSAT [1]. We do not use CADICAL [2] as [13], because PICOSAT allows to directly generate a resolution proof in the TRACECHECK format. The TRACECHECK proof alongside with the original CNF is passed on to our tool DRUP2PAC. In DRUP2PAC we encode the resolution proof as a PAC proof, by re-enacting the resolution steps in the given traces using algebraic reasoning. However we do not want to derive the empty clause, as this corresponds to deriving the constant polynomial 1. Hence whenever we encounter the unit clause encoding the assumption of the miter in a trace, we remove it from the trace.

As a further optimization we internally apply *bit-flipping*, as for instance proposed in [17], on the algebraic level to keep the size of the intermediate polynomials small. It can be seen in the polynomial encoding of the CNF in Fig. 1, that each positive literal l in a clause introduces a factor $(1-l)$ in the corresponding polynomial encoding of the clause. As the PAC format uses only the expanded form of polynomials, expanding clauses with multiple positive literals leads to a tremendous growth in the polynomial encoding. In order to overcome this issue, we introduce for each literal l_i a bit-flipping polynomial $-f_i - l_i + 1 = 0$ in the constraints set and internally flip variables in the CNF such that only negative literals are contained. We monitor the bit-flipping in the clausal polynomials by generating corresponding PAC rules and add the bit-flipping polynomials to the constraints set.

After translating the full TRACECHECK proof we derive for each pair of miter inputs the equations $s_i(s'_i - 1) = 0$ and $s'_i(s_i - 1) = 0$ using unit propagation. Encoding unit

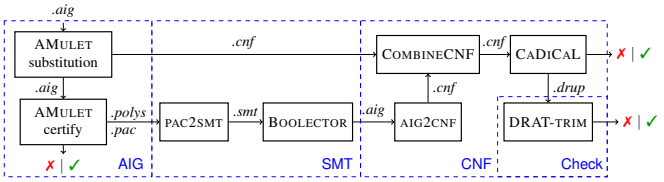


Fig. 3. Converting a PAC proof into a DRUP proof

propagation in PAC is very similar to encoding resolution. Subtracting these polynomials leads to $s_i - s'_i = 0$ for each pair of input bits. We report these pairs in ".equal".

We certify the rewritten AIG using AMULET. At this point the specification which we derived in the PAC proof uses the outputs s'_i of the RCA. Our final tool MATCHSPEC generates PAC rules which replace every occurrence of s'_i by the corresponding bit s_i using the equations $s_i - s'_i = 0$. As a last step the generated proof and the original specification (in terms of s_i) is checked using PACTRIM.

IV. FROM PAC TO DRUP

We have seen how to encode a DRUP proof into PAC. However, not only is PAC more complex than DRUP (and DRAT), but PAC neither has certified proof checkers, while DRAT and thus DRUP can be translated to LRAT, for which such checkers exist [6]. Therefore it is natural to ask, whether it is possible to translate PAC proofs into DRUP.

In this section we give a positive but impractical answer. The first hurdle is to encode the specification into CNF. This can in principle be achieved using SMT over the theory of bit-vectors for a large enough bit-width followed by bit-blasting. However, at this point, we are not able to track rewriting within SMT solvers, which leaves a gap in the proof. A further issue of our encoding is that we only translate each PAC rule individually to SMT and CNF. We do not include a check that the specification of the circuit is derived at the end, which is another gap in our proof. Thus our resulting DRUP proof is far from being a complete proof, in the sense of covering every rewriting step. The size of these proofs can only be considered as an empirically derived lower bound.

Note that our translation introduces new variables and thus technically needs extended resolution (ER), thus actually DRAT. But we continue to use DRUP instead of DRAT to describe our approach. Our tool flow can be seen in Fig 3. We apply adder substitution and certify the rewritten multiplier as in [13]. In our tool PAC2SMT we abstract the polynomial proof to a bit-vector proof. To this end we encode the PAC proof as an SMT problem over the theory of quantifier free fixed size bit vectors. Note that each variable in the PAC proof represents the input or output of a gate. As a consequence we encode each variable in the PAC proof as a single bit and the coefficients are encoded as bit vectors. The length of the bit vectors depends on the highest coefficient in the PAC proof.

In our encoding of the PAC rules we include the following optimization when single bits are multiplied with bit-vectors.

```
(bvand #b011 ((_ sign_extend 2) x)) =
(bvmul #b011 ((_ zero_extend 2) x)).
```

We encode each rule of the PAC proof as a bit vector equation and assume that the conjunction of all these equations is unsatisfiable. The SMT encoding is given to BOOLECTOR [15], which additionally is able to generate AIGs from bit vector formulas. As discussed above, BOOLECTOR applies rewriting steps, which are not covered in the DRUP proof. Using the tool AIG2CNF from the Aiger library [3] we translate the AIG into a CNF. Nodes from AIGs can easily be encoded in CNF, as indicated in Fig. 1.

At this point we have two CNF encodings. The first CNF is directly produced by AMULET and encodes the bit-level miter proving the correctness of the adder substitution. The second CNF encodes the translated PAC proof. Both CNFs are encoded to deliver a refutation, i.e., for a correct multiplier both CNFs should be unsatisfiable. More precisely each CNF encodes a miter, thus both CNF contain one unit clause $C_i = l_i$ which represents the assumption for the miter output.

The CNFs are merged by collecting all clauses, except the clause encoding the output assumption. The two output clauses $C_0 = l_0, C_1 = l_1$ are merged into the clause $l_0 \vee l_1$, thus either l_0 or l_1 needs to be true to satisfy the CNF. As we expect that both l_0 and l_1 are false, the clause $l_0 \vee l_1$ should be unsatisfiable, and thus the whole CNF too. The merged CNF is solved using the SAT solver CADICAL [2], which is instructed to generate a DRUP proof. Finally this proof is checked using DRAT-TRIM [18].

V. EXPERIMENTS

Our experiments were conducted on Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (with turbo-mode disabled). The time in Tbl. I is listed in rounded seconds (wall-clock time) and we measure the time from starting the tools until they finished or an error occurred, e.g. the time limit was reached, set to 3600 sec (1h), or the memory limit of 128 GB.

We consider various multiplier architectures used in our experiments in [13]. Benchmarks are generated with the Arithmetic Module Generator [11] and BOOLECTOR [15]. Except for the "btor" benchmarks from BOOLECTOR, the selected architectures contain generate-and-propagate final stage adders, thus adder substitution was required and applied as in [13] and hence a DRUP as well as a PAC proof were generated. These proofs are translated as explained in Sect. III and Sect. IV.

The first block shows the time for generating and checking the proofs as in [13]. For "DRUP" and "PAC" we present the time it takes to generate the corresponding proof and the time to check proofs using DRAT-TRIM [18] and PACTRIM [16]. Additionally we depict the sizes of the proofs. The proof size of DRUP proofs is measured by the number of added RUP clauses [10]. The size of PAC proofs is defined by the number of applied PAC rules [16].

The second block "PAC" shows the time for generating a single PAC proof as described in Sect. III. Almost all of the time in proof generation is used by converting the DRUP proof to a PAC proof, eg. for "bp-wt-cl-32" our tool DRUP2PAC needs 3130 seconds. We are able to generate and check PAC proofs up to bit-width 32. The growth in the proof size depends

TABLE I
PROOF GENERATION AND CHECKING

architecture	n	[13] DRUP gen	[13] DRUP check	[13] DRUP size	[13] PAC gen	[13] PAC check	[13] PAC size	[13] total	PAC gen	PAC check	PAC total	PAC size	DRUP aig	DRUP smt	DRUP cnf	DRUP check	DRUP total	DRUP size
btor	16	-	-	-	0	0	5 181	0	-	-	-	-	0	3	136	177	316	11 079 431
sp-ar-cl	16	0	0	1 299	0	0	7 962	0	2	2	3	185 588	0	7	300	264	570	19 317 884
sp-dt-lf	16	0	0	1 167	0	0	7 787	0	1	1	2	136 349	0	6	279	277	562	18 153 668
bp-ct-bk	16	0	0	1 029	0	0	7 205	0	1	1	2	128 720	0	7	TO	-	-	-
bp-wt-cl	16	0	0	2 902	0	0	7 946	0	30	11	41	614 742	0	7	TO	-	-	-
btor	32	-	-	-	0	0	21 629	0	-	-	-	-	0	32	2 887	TO	-	-
sp-ar-cl	32	0	0	14 927	0	1	33 834	1	133	31	164	1 597 897	0	56	TO	-	-	-
sp-dt-lf	32	0	0	3 138	0	1	33 451	1	2	3	5	321 720	0	52	TO	-	-	-
bp-ct-bk	32	0	0	2 276	0	1	27 312	1	1	2	3	217 128	0	49	TO	-	-	-
bp-wt-cl	32	1	1	46 502	0	1	30 561	2	3 133	242	3 375	5 536 176	0	55	TO	-	-	-

PPG: simple (sp), Booth (bp) PPA: array (ar), Dadda tree (dt), compressor tree (ct), Wallace tree (wt) TO = 3600 sec
FSA: carry look-ahead (cl), Ladner-Fischer (lf), Brent-Kung (bk) Benchmarks are generated by the Arithmetic Module Generator [11].

highly on the benchmark, more precisely it depends on the generated DRUP proof. For instance for 32 bit, the increment of the PAC proofs is between factor 10 and factor 1600.

The third block "DRUP" lists the time for generating and checking a single DRUP proof as described in Sect. IV. Column "aig" shows the time needed for adder substitution and generating the PAC proof. In column "smt" we present the time needed to generate an SMT proof as well as the time BOOLECTOR [15] needs to generate a CNF out of the SMT proof. The following column "cnf" lists the time we need to combine and solve the CNFs using CADICAL. We are only able to generate and check DRUP proofs up to 16 bit. The size of the DRUP proofs compared to the single PAC proofs increases drastically. Especially for the "btor" benchmarks, where no initial DRUP proof is generated, converting the PAC proof to the DRUP proof increases the size by three orders of magnitude. These are still not complete DRUP proofs. Neither rewriting, nor the extensions to encode bit-blasting (both requiring DRAT) are accounted yet.

VI. CONCLUSION

State-of-the-art verification techniques of arithmetic circuits rely on SAT as well as computer algebra. However they lack a proof certificate in a single proof format. With two proof formats we argue that additional manual compositional reasoning would be required to certify the verification. In this paper we present how to translate the clausal reasoning proof format DRUP into the algebraic proof format PAC and vice versa in order to produce one single proof certificate.

Translating DRUP proofs to PAC proofs requires algebraic reasoning. We include bit-flipping techniques in order to reduce the size of polynomials. As a further optimization we use the TRACECHECK format as input format, in order to directly determine the necessary polynomial equations.

To obtain DRUP from PAC proofs we encode the PAC proofs as an SMT problem, which then is translated into CNF using bit-blasting by an SMT solver. However, this intermediate step leaves gaps in the proof, since we are not able to track internals of SMT solving. Even though far from being complete proofs, they serve as empirically derived lower bounds on such clausal proofs. These proofs are three orders of magnitude larger than the corresponding PAC proofs.

As future work we want to be able to close the gap in generating DRUP proofs. Generating smaller proofs by applying more sophisticated reasoning is interesting as well.

REFERENCES

[1] A. Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.
[2] A. Biere. CaDiCaL at the SAT Race 2019. In *Proc. of SAT Race 2019*, 2019. Submitted.
[3] A. Biere, K. Heljanko, and S. Wieringa. AIGER 1.9 And Beyond. Technical report, FMV Reports Series, JKU Linz, Austria, 2011.
[4] M. Ciesielski, T. Su, A. Yasin, and C. Yu. Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model. *IEEE TCAD*, pages 1–1, 2019.
[5] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms*. Springer-Verlag New York, 1997.
[6] L. Cruz-Filipe, J. Marques-Silva, and P. Schneider-Kamp. Formally Verifying the Solution to the Boolean Pythagorean Triples Problem. *J. Autom. Reasoning*, 63(3):695–722, 2019.
[7] A. V. Gelder. Verifying RUP proofs of propositional unsatisfiability. In *ISAIM*, 2008.
[8] A. V. Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.
[9] M. Heule, W. A. Hunt Jr., and N. Wetzler. Trimming while checking clausal proofs. In *FMCAD 2013*, pages 181–188. IEEE, 2013.
[10] M. J. H. Heule and A. Biere. Proofs for Satisfiability Problems. In *All about Proofs, Proofs for All*, volume 55, pages 1–22, 2015.
[11] N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi. Formal Design of Arithmetic Circuits Based on Arithmetic Description Language. *IEICE Transactions*, 89-A(12):3500–3509, 2006.
[12] W. A. Hunt, M. Kaufmann, J. Strother Moore, and A. Slobodova. Industrial hardware and software verification with ACL2. *Philos. Trans. Royal Soc. A*, 375:20150399, 10 2017.
[13] D. Kaufmann, A. Biere, and M. Kauers. Verifying Large Multipliers by Combining SAT and Computer Algebra. In *FMCAD 2019*, pages 28–36. IEEE, 2019.
[14] A. Mahzoon, D. Große, and R. Drechsler. RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. In *Design Automation Conf.*, 2019. In press.
[15] A. Niemetz, M. Preiner, C. Wolf, and A. Biere. Btor2 , BtorMC and Boolector 3.0. In *Computer Aided Verification, CAV*, volume 10981 of *LNCS*, pages 587–595. Springer, 2018.
[16] D. Ritirc, A. Biere, and M. Kauers. A Practical Polynomial Calculus for Arithmetic Circuit Verification. In *SC-Square Workshop 2018*, pages 61–76. CEUR-WS, 2018.
[17] A. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler. Equivalence checking using Gröbner bases. In *FMCAD*, pages 169–176. IEEE, 2016.
[18] N. Wetzler, M. Heule, and W. A. Hunt Jr. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *SAT*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014.