Formal Methods in Computer-Aided Design 2020

The Proof Checkers Pacheck and Pastèque for the Practical Algebraic Calculus

Daniela Kaufmann Mathias Fleury Armin Biere
Johannes Kepler University Linz, Altenbergerstr. 69, 4040 Linz, Austria
daniela.kaufmann@jku.at mathias.fleury@jku.at armin.biere@jku.at

***Abstract*—Generating and checking proof certificates is important to increase the trust in automated reasoning tools. In recent years formal verification using computer algebra became more important and is heavily used in automated circuit verification. An existing proof format which covers algebraic reasoning and allows efficient proof checking is the practical algebraic calculus. In this paper we present two independent proof checkers PACHECK and PASTÈQUE. The checker PACHECK checks algebraic proofs more efficiently than PASTÈQUE, but the latter is formally verified using the proof assistant Isabelle/HOL. 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 rules too.**

I. Introduction

Formal verification aims to guarantee the correctness of a given system with respect to a certain specification. However, the verification process might contain errors. In order to increase the trust in verification results, it is common to generate proof certificates, which can be checked by a stand-alone proof checker. For example, in the SAT competition certificates of unsatisfiability are required since 2013 and different resolution and clausal proof formats [1], such as DRUP [2], [3], DRAT [4], and LRAT [5] are available.

Automated reasoning based on computer algebra has a long history [6]–[8] with renewed recent interest; e.g., it provides the state of the art in verifying gate-level multipliers [9]–[12]. Furthermore, algebraic reasoning in combination with satisfiability checking (SAT) is succesfully used to solve complex combinatorial problems [13]–[16] with possible future applications in cryptanalysis [17]–[19].

The practical algebraic calculus (PAC) [20] is a proof format to represent certificates for validating results of such algebraic techniques. It is based on the polynomial calculus (PC) [21] and allows to dynamically capture that a polynomial can be derived from a given set of polynomials using algebraic ideal theory. In contrast to PC, PAC proofs can be checked efficiently, for example using our tool PACTRIM [20].

In this paper we add an indexing scheme to PAC and also propose deletion and extension rules. Our paper contains no new theory, except for the more technical formalization of extensions. This allows us to merge and check proofs obtained from SAT and computer algebra [22], the current state-of-the-art, in a uniform (and now precise) manner. The purpose of this system description is to define the new version of PAC and present our new checkers PACHECK and PASTÈQUE. Furthermore, PASTÈQUE in contrast to PACHECK is verified in Isabelle/HOL, but PACHECK is faster and more memory efficient (also compared to PACTRIM). A preliminary version of this paper is included in the first author's PhD thesis [23].

II. Practical Algebraic Calculus

In this section we briefly introduce the algebraic notion following [24]. Let X be the set of variables $\{x_1, \ldots, x_n\}$ and further let $G \subseteq \mathbb{Z}[X]$ and $f \in \mathbb{Z}[X]$.

Algebraic proof systems reason about polynomial equations. The aim is to show that the equation $f = 0$ is implied by the equations $g = 0$ for every $g \in G$; i.e., every common root of the polynomials $g \in G$ is also a root of f. In algebraic terms, this question means to derive whether f belongs to the ideal generated by G. A nonempty subset $I \subseteq \mathbb{Z}[X]$ is called an *ideal* if $\forall u, v \in I : u+v \in I$ and $\forall w \in \mathbb{Z}[X], \forall u \in I : wu \in I$. If $G = \{g_1, \ldots, g_m\} \subseteq \mathbb{Z}[X]$, then the ideal generated by G is defined as $\langle G \rangle = \{q_1 g_1 + \cdots + q_m g_m \mid q_1, \ldots, q_m \in \mathbb{Z}[X]\}$.

For a given set of polynomials $G \subseteq \mathbb{Z}[X]$, a *model* is a point $u = (u_1, \ldots, u_n) \in \mathbb{Z}^n$ such that $\forall g \in G : g(u_1, \ldots, u_n) = 0$. Here, by $g(u_1, \ldots, u_n)$ we mean the element of $\mathbb{Z}$ obtained by evaluating the polynomial g for $x_1 = u_1, \ldots, x_n = u_n$.

PAC proofs [20] are sequences of proof rules. 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 determine that P at index i is set to p.

The initial state is $(X = \text{Var}(G \cup \{f\}), P)$ where P maps indices to polynomials of G. For bit-level verification [20] only models of the Boolean domain $\{0, 1\}^n$ are of interest. In previous work, we added the set of Boolean-value constraints $B(X) = \{x^2 - x \mid x \in X\}$ to G and had 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, when checking the correctness, we immediately reduce exponents greater than one in the polynomials. The following two rules model the properties of ideals as introduced above.

[ADD (i, j, k, p)] $(X, P) \implies (X, P(i \mapsto p))$
  provided that $P(j) \neq \bot$, $P(k) \neq \bot$, $P(i) = \bot$, $p \in \mathbb{Z}[X]/\langle B(X) \rangle$, and $p = P(j) + P(k) \mod \langle B(X) \rangle$.

[MULT (i, j, q, p)] $(X, P) \implies (X, P(i \mapsto p))$
  provided $P(j) \neq \bot$, $P(i) = \bot$, $p, q \in \mathbb{Z}[X]/\langle B(X) \rangle$, and $p = q \cdot P(j) \mod \langle B(X) \rangle$.

```
     letter  ::=  'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
     number  ::=  '0' | '1' | ... | '9'
   constant  ::=  (number)+
   variable  ::=  letter (letter | number)*
       term  ::=  variable ('*' variable)*
   monomial  ::=  constant |  [constant '*'] term
 polynomial  ::=  ['-'] monomial ('+' | '-' monomial)*
      index  ::=  constant
      input  ::=  (index polynomial ';')*
   add_rule  ::=  index '+' index ',' index ',' polynomial ';'
   mul_rule  ::=  index '*' index ',' polynomial ',' polynomial ';'
   del_rule  ::=  index 'd' ';'
   ext_rule  ::=  index '=' variable ',' polynomial ';'
      proof  ::=  (add_rule | mul_rule | del_rule | ext_rule)*
     target  ::=  polynomial ';'
```

Figure 1. Syntax of input polynomials, target, and proofs in PAC-format

If in either one of the above rules p is also the target polynomial f, it holds that $f \in \langle G \rangle$. In the original PAC format introduced in [20], it was necessary to explicitly provide the antecedents $P(i)$ and $P(j)$. In our new format, we use indices i and j to access polynomials, similar to LRAT [5]. The new syntax is given in Fig. 1 and an example is provided with our tools [25]. Naming polynomials by indices reduces the proof size and makes parsing more efficient, because only the conclusion polynomials of each rule and the initial polynomials of G are stated explicitly. However, introducing indices for polynomials has the effect that the semantics of P changes from sets to multisets, as in DRAT [3], and it is possible to introduce the same polynomial under different names. Checking the result of each rule allows pinpointing the first error, instead of claiming that the proof is wrong somewhere in one of the (usually millions) of steps.

We extend our original proof rules [20] by adding a deletion and an extension rule. 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.

$$[\text{DELETE}(i)] \qquad (X, P) \implies (X, P(i \mapsto \bot))$$

A. *Extension*

In our previous work [22], we converted DRUP proofs to the PAC format and encountered the need to extend the initial set of polynomials G to reduce the size of the polynomials in the PAC proof. We included polynomials of the form $-f_x + 1 - x$, similar to the negation rule in the polynomial calculus with resolution [26], which introduced the variable f_x as the negation of the Boolean variable x.

However, at that point we did not use proper extension rules, but simply added these extension polynomials to the initial polynomials G. This may affect the models of the constraint set, because any arbitrary polynomial can be added as initial constraints. 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 to add further polynomials to the knowledge base with new variables while preserving the original models on the original variable set of variables X.

$$[\text{EXT}\ (i, v, p)] \qquad (X, P) \implies (X \cup \{v\}, P(i \mapsto -v + p))$$

provided that $P(i) = \bot$ and $v \notin X$ and $p \in \mathbb{Z}[X]/\langle B(X)\rangle$, and $p^2 - p \equiv 0 \mod \langle B(X)\rangle$.

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 $p^2 - p \equiv 0 \mod \langle B(X)\rangle$. Without this condition v might take non-Boolean solutions and thus force us to calculate in the ring $\mathbb{Z}[X, v]/\langle B(X)\rangle$ instead of $\mathbb{Z}[X, v]/\langle B(X, v)\rangle$.

Consider for example $P = \{-y + x - 1\}$. The only Boolean model is $(x, y) = (1, 0)$. If we extend P by $-v + x + 1$ we derive $v = 2$, because $x = 1$ for all models of P. 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 $P \subseteq \mathbb{Z}[X]/\langle B(X)\rangle$. Let "$<$" be a lexicographic ordering, H a Gröbner basis [27] of $\langle P \rangle$ w.r.t. "$<$", and "$<_v$" be an extension of "$<$" with v as largest element. Thm. 3 of [28] shows that $H \cup \{p_v\}$ is a Gröbner basis w.r.t. "$<_v$" for $\langle P_v \rangle := \langle P(i \mapsto p_v)\rangle \subseteq \mathbb{Z}[X \cup \{v\}]/\langle B(X \cup \{v\})\rangle$, the extended ideal, and $\langle P_v \rangle \cap \mathbb{Z}[X]/\langle B(X)\rangle = \langle H \cup \{p_v\}\rangle \cap \mathbb{Z}[X]/\langle B(X)\rangle = \langle H \rangle = \langle P \rangle$ follows. $\square$

III. PACHECK

We implemented PACHECK as an extension of our previous checker PACTRIM [20]. It consists of approximately 1 700 lines of C code and is published [25] under MIT license. The default mode of PACHECK supports the extended version of PAC, as presented in this paper, for the new syntax using indices. PACHECK is backwards compatible to our original format of PAC [20] and all features including reasoning with exponents are supported. However, extension rules are only supported for Boolean models.

PACHECK reads three input files `<input>`, `<proof>`, and `<target>` and then verifies that the polynomial in `<target>` is contained in the ideal generated by the polynomials in `<input>` using the rules provided in `<proof>`. The polynomial arithmetic needed for checking the proof rules is implemented from scratch. In PACHECK polynomials are stored as ordered linked lists of monomials, where a monomial consists of a coefficient and a term. The coefficients are represented using the GMP library [29]. Terms are ordered linked list of variables that are identified as strings.

In the default mode of PACHECK we order variables in terms lexicographically using `strcmp`. All internally allocated terms in linked lists are shared using a hash table. It turns out that the order of variables has an enormous effect on memory usage, since different variable orderings induce different terms. For example, given the monomials xyz and $x'yz$, sharing of yz is possible for the order $x' > x > y > z$, whereas no sharing occurs for $y > x > z > x'$. 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 same variable ordering as in the given proof files. That is, we assign increasing `level` values to new variables 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 rules. PACHECK supports the orderings `strcmp`, `-1*strcmp`, `level`, and `-1*level`. Terms in polynomials are sorted using the same order as for the variables.

In the initial phase of PACHECK each polynomial from `<input>` is sorted and stored as an inference. Inferences consist of a given index and a polynomial and are stored in a hash table. In the default mode, the index acts as the hash value. Thus it is possible to add the same polynomial twice. If the original format of PAC is used, a hash value is computed based on the input polynomial. Proof checking is applied on-the-fly. We parse each rule of `<proof>` and immediately apply the necessary checks discussed in Sect. II. If the rule is either ADD or MULT, we have to compute whether the conclusion polynomial of the rule is equal to the arithmetic operation performed on the antecedent polynomials.

We modified the algorithm of polynomial addition in PACTRIM and now assume the monomials of polynomials to be sorted. Addition of polynomials is performed by merging their monomials in an interleaved way. In PACTRIM we pushed the monomials of both polynomials on a stack and then sorted and merged them. Normalization of the exponents is not necessary in the ADD rule, but we still use this technique for multiplication of polynomials, where we multiply each monomial of the first polynomial with each monomial of the second monomial. In the MULT rule we normalize exponents larger than one, before testing equality. Furthermore, we check whether the conclusion polynomial of the rules ADD or MULT matches the polynomial in `<target>` in order to identify whether the target polynomial was derived.

The original version of PACTRIM [20] did not allow to delete inferences. As a consequence the set of polynomials increased with each proof rule, leading to memory exhaustion for very large proofs. In PACHECK we now support deletion of inferences. A partial solution for deletion was used in [9] to reduce memory usage. However, in contrast to our new version, individual inferences could not be deleted (only both antecedents of a proof step could be). Extension variables were not supported in PACTRIM [20] either.

IV. PASTÈQUE

To further increase trust in the verification, we implemented a verified checker called PASTÈQUE in the proof assistant Isabelle/HOL [30]. It follows a "refinement" approach, starting with an abstract specification of ideals, which we then refine with the Isabelle Refinement Framework [31] to the transition system from Sect. II, and further down to executable code using Isabelle's code generator [32]. The Isabelle files have been made available [33]. The generated code consists of 2 800 lines Standard ML (2 400 generated by Isabelle, 400 for the parser) and is also available [25] 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 rules. 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., neither "$\mathrm{Var}\,(1) = \emptyset$" nor "$p \neq 0 \implies X \in \mathrm{Var}\,(X \times p)$" are present). However, Sledgehammer [34] automatically proved many of these simple lemmas.

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 [35] 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 parser and can be compiled using the Standard ML compiler MLTON [36].

The implementation is not backwards compatible and 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 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. While Isabelle's code generator to LLVM [37] 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. 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 are verified, not the main loop.

V. TOOL DEMONSTRATION

In this section we show an example of a PAC proof and the output of our new checkers, which demonstrates the usage of our tools PACHECK and PASTÈQUE.

Example 1. *Let $\bar{x} \vee \bar{y}$ and $y \vee z$ be two clauses. From these clauses we are able to derive the clause $\bar{x} \vee z$ using resolution. We show how this derivation can be covered in PAC.*

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 $xy = 0$.

We translate the given clauses, which builds our input `<res.input>` *and the target* `<res.target>`*. For the PAC proof in* `<res.proof>` *we introduce an extension variable f_z, which models the negation of z, i.e. $-f_z + 1 - z = 0$. We use this extension to reduce the size of the conclusion polynomials. The PAC proof shows only some possible deletion rules, adding more deletion rules is possible. The files of this example are available [25].*

```
<res.input>        <res.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;
                    2  d;
                    4  d;
<res.target>        6  *  1,   fz, fz*x*y;
 -x*z+x;            1  d;
                    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;
```

We give these files to PACHECK *and* PASTÈQUE *and the results are provided in the Figs 2 and 3.*

```
$ pacheck res.input res.proof res.target
[pacheck] Pacheck Version 001
[pacheck] Practical Algebraic Calculus Proof Checker
[pacheck] Copyright (C) 2020, Daniela Kaufmann, JKU
[pacheck] compressed mode with indices assumed
[pacheck] checking target enabled
[pacheck] reading target polynomial from 'res.target'
[pacheck] read 8 bytes from 'res.target'
[pacheck] reading original polynomials from 'res.input'
[pacheck] found 2 original polynomials in 'res.input'
[pacheck] read 20 bytes from 'res.input'
[pacheck] reading polynomial algebraic calculus proof from
↪  'res.proof'
[pacheck] found and checked 8 inferences in 'res.proof'
[pacheck] read 219 bytes from 'res.proof'
[pacheck] found 1 target polynomial inference
[pacheck] proof length 10 (number of polynomials)
[pacheck] proof size 25 (on average 2.5 terms per
↪  polynomial)
[pacheck] proof degree 3 (internal maximum degree 3)
[pacheck] searched 32 inferences 0.1 average collisions
[pacheck] 10 inferences, 3.2 average searches
[pacheck] original inferences 2 (20% of total rules)
[pacheck] inference rules 8 (80% of total rules)
[pacheck] addition inference rules 3 (38% of inference
↪  rules)
[pacheck] multiplication inference rules 4 (50% of inference
↪  rules)
[pacheck] extension rules 1 (12% of inference rules)
[pacheck] deletion inference rules 3 (30% of total rules)
[pacheck] maximum 9 of total 10 terms (90%)
[pacheck] searched 52 terms 81% hits 0.3 average collisions
[pacheck] maximum 2229 bytes allocated (0.0 MB)
[pacheck] maximum resident set size 4481024 bytes (4.3 MB)
[pacheck] process time 0.000 seconds
[pacheck] TARGET CHECKED
```

Figure 2. Output of PACHECK on the example from Ex. 1.

```
$ pasteque res.input res.proof res.target
c polys parsed
c ******************
c pac parsed
c spec parsed
c Now checking
s SUCCESSFULL
c
c ***** stats *****
c parsing polys file init (nonGC): 0.000 s = 0.000 s (usr)
↪  0.000 s (sys)
c parsing pac file init (nonGC): 0.000 s = 0.000 s (usr)
↪  0.000 s (sys)
c full init (nonGC): 0.000 s = 0.000 s (usr) 0.000 s (sys)
c time solving (nonGC): 0.000 s = 0.000 s (usr) 0.000 s
↪  (sys)
c time GC: 0.000 s = 0.000 s (usr) 0.000 s (sys)
c time solving(full): 0.000 s
c Overall (nonGC): 0.001 s = 0.001 s (usr) 0.000 s (sys)
c overall GC: 0.000 s = 0.000 s (usr) 0.000 s (sys)
c Overall(full): 0.001 s
```

Figure 3. Output of PASTÈQUE on the example from Ex. 1.

VI. Evaluation

In our experiments we used 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 highlight the benefits of the new proof format and provide a comprehensive comparison between our two tools. Source code, benchmarks and experimental data are available [25].

For the experiments of Table I we generated PAC proofs as in previous work [9], [22] to validate the correctness of multipliers with input bit-width n. The circuits are either generated with AMG [38], BOOLECTOR [39] or GENMUL [40].

For the upper part of Table I we generated proof certificates with AMULET [9] to validate the correctness of simple multiplier circuits [9]. We modified AMULET to generate proofs in our new PAC format.

Our previous approach [9] 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 [22]. In the experiments of [22] all gate constraints 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 (cf. Sect. II-A). All these polynomials are now added using extension rules. This preserves the models of the gate constraints of the given multiplier. Experiments for these proof certificates are shown in the lower part of Table I. The second column shows the input bit-width, the third column shows the number of generated proof steps and the fourth the highest degree of the polynomials.

Table I
PROOF CHECKING (IN BOLD THE FASTEST VERSION)

multiplier	n	steps (10^6)	deg	PACTRIM		PACHECK						PASTÈQUE			
						no delete		no index		default		default		`uloop`	
				sec	MB	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB
btor	128	0.4	3	10	105	**5**	273	11	100	**5**	92	22	3 886	17	1 773
btor	256	1.6	3	60	459	**25**	1 144	62	435	**25**	364	105	21 157	79	4 364
btor	512	6.3	3	395	2 066	**138**	4 956	402	1 972	141	1 461	531	64 412	416	22 292
sp-ar-rc	128	0.6	4	16	156	**6**	454	16	148	**6**	136	31	5 002	23	1 608
sp-ar-rc	256	2.3	4	92	687	29	1 858	96	651	**27**	541	139	32 525	102	8 769
sp-ar-rc	512	9.4	4	587	3 107	146	7 683	617	2 965	**134**	2 171	608	64 412	471	25 632
sp-ar-cl	32	1.6	256	31	405	23	773	36	354	**21**	353	121	40 654	113	9 492
sp-dt-lf	32	0.3	46	3	82	**2**	122	3	73	**2**	73	11	1 679	11	886
bp-ct-bk	32	0.2	25	2	57	**1**	86	2	52	**1**	51	8	1 600	7	1 068
bp-wt-cl	32	5.6	764	242	1 716	193	4 324	302	1 430	**181**	1 428	786	58 867	774	64 404

The columns PACTRIM show the time and memory usage of our previous proof checker PACTRIM. For that we reproduce proofs of [9], [22] in the original PAC format. These proofs are also used in the column "no index" to show the backward compatibility of PACHECK. It can be seen that PACTRIM and PACHECK behave similar on the original PAC format.

The effect of deletion rules and indices in PACHECK can also be seen in Table I. Deletion rules reduce the memory usage by at least a factor two, although the effect on runtime is limited. Using indices reduces the runtime by 30 to 80%. Note that in our earlier experiments [22] 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 $p \in \mathbb{Z}[X]/\langle B(X)\rangle$ and $p^2 - p \equiv 0 \mod \langle B(X)\rangle$.

Furthermore, we can compare the performance of PACHECK and PASTÈQUE. The memory usage for PASTÈQUE depends on the garbage collector, which likely explains the peak around 64 GB (half of the available memory). The verified checker PASTÈQUE is less efficient. It is both much slower and more memory hungry. Verified checkers of SAT certificates [41], [42] have the same level of efficiency as state-of-the-art checkers [43], likely because of the imperative style (unlike our pure functional code) and the more efficient memory usage by managing most memory directly (e.g., for clauses) instead of relying on the garbage collector.

VII. CONCLUSION AND FUTURE WORK

We presented our proof checkers PACHECK and PASTÈQUE which are able to check PAC proofs efficiently. Our new proof format includes an extension rule, which is able to capture rewriting techniques. Furthermore, we added a deletion rule and used indices for polynomials. Our experiments showed that these optimizations cut memory usage in half and reduce the runtime by around 30–80%. PACHECK was four 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 II 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 the newest version of our tools [9] 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.

Acknowledgement: This work is supported by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), and LIT AI Lab funded by the State of Upper Austria.

REFERENCES

[1] M. J. H. Heule and A. Biere, "Proofs for satisfiability problems," in *All about Proofs, Proofs for All*, vol. 55, 2015, pp. 1–22.

[2] A. Van Gelder, "Verifying RUP proofs of propositional unsatisfiability," in *ISAIM*, 2008.

[3] ——, "Producing and verifying extremely large propositional refutations – have your cake and eat it too," *Ann. Math. Artif. Intell.*, vol. 65, no. 4, pp. 329–372, 2012.

[4] M. J. H. Heule, W. A. Hunt, Jr., and N. Wetzler, "Trimming while checking clausal proofs," in *Formal Methods in Computer-Aided Design, FMCAD 2013*. IEEE, 2013, pp. 181–188. [Online]. Available: http://ieeexplore.ieee.org/document/6679408/

[5] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt, Jr., M. Kaufmann, and P. Schneider-Kamp, "Efficient certified RAT verification," in *CADE 26*, ser. LNCS, L. de Moura, Ed., vol. 10395. Springer, 2017, pp. 220–236.

[6] D. Kapur, "Geometry theorem proving using Hilbert's Nullstellensatz," in *SYMSAC*. ACM, 1986, pp. 202–208.

[7] ——, "Using Gröbner bases to reason about geometry problems," *J. Symb. Comput.*, vol. 2, no. 4, pp. 399–408, 1986.

[8] D. Kapur and P. Narendran, "An equational approach to theorem proving in first-order predicate calculus," in *IJCAI*. Morgan Kaufmann, 1985, pp. 1146–1153.

[9] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *FMCAD 2019*. IEEE, 2019, pp. 28–36.

[10] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *DAC*. ACM, 2019, pp. 185:1–185:6.

[11] M. J. Ciesielski, T. Su, A. Yasin, and C. Yu, "Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model," *IEEE TCAD*, pp. 1–1, 2019.

[12] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *DATE 2020*. IEEE, 2020, pp. 544–549.

[13] C. Bright, I. Kotsireas, and V. Ganesh, "Applying computer algebra systems and SAT solvers to the Williamson conjecture," *Journal of Symbolic Computation*, 04 2018.

[14] M. J. H. Heule, "Computing small unit-distance graphs with chromatic number 5," *CoRR*, vol. abs/1805.12181, 2018.

[15] M. J. H. Heule, M. Kauers, and M. Seidl, "Local search for fast matrix multiplication," in *SAT 2019*, ser. LNCS, vol. 11628. Springer, 2019, pp. 155–163.

[16] ——, "New ways to multiply 3×3-matrices," *CoRR*, vol. abs/1905.10192, 2019.

[17] C. Condrat and P. Kalla, "A gröbner basis approach to cnf-formulae preprocessing," in *TACAS 2007*, ser. LNCS, vol. 4424. Springer, 2007, pp. 618–631.

[18] M. Soos and K. S. Meel, "BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting," in *AAAI 2019*. AAAI Press, 2019, pp. 1592–1599.

[19] D. Choo, M. Soos, K. M. A. Chai, and K. S. Meel, "Bosphorus: Bridging ANF and CNF solvers," in *DATE 2019*, J. Teich and F. Fummi, Eds. IEEE, 2019, pp. 468–473.

[20] D. Ritirc, A. Biere, and M. Kauers, "A practical polynomial calculus for arithmetic circuit verification," in *SC2'18*, A. Bigatti and M. Brain, Eds. CEUR-WS, 2018, pp. 61–76.

[21] M. Clegg, J. Edmonds, and R. Impagliazzo, "Using the Groebner basis algorithm to find proofs of unsatisfiability," in *STOC*. ACM, 1996, pp. 174–183.

[22] D. Kaufmann, A. Biere, and M. Kauers, "From DRUP to PAC and back," in *DATE 2020*. IEEE, 2020, pp. 654–657. [Online]. Available: http://fmv.jku.at/drup2pac/

[23] D. Kaufmann, "Formal verification of multiplier circuits using computer algebra," Ph.D. dissertation, Informatik, Johannes Kepler University Linz, 2020.

[24] D. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms*. Springer-Verlag New York, 1997.

[25] D. Kaufmann and M. Fleury, "The PAC checkers Pacheck and Pastèque," accessed: 2020-05-06. [Online]. Available: http://fmv.jku.at/pacheck_pasteque

[26] M. Alekhnovich, E. Ben-Sasson, A. A. Razborov, and A. Wigderson, "Space complexity in propositional calculus," *SIAM J. Comput.*, vol. 31, no. 4, pp. 1184–1211, 2002.

[27] B. Buchberger, "Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal," Ph.D. dissertation, University of Innsbruck, 1965.

[28] D. Lichtblau, "Effective computation of strong Gröbner bases over Euclidean domains," *Illinois Journal of Mathematics*, vol. 56, 11 2013.

[29] T. Granlund *et al.*, "GNU Multiple Precision Arithmetic Library," January 2020. [Online]. Available: http://gmplib.org/

[30] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[31] P. Lammich, "Refinement based verification of imperative data structures," in *CPP 2016*, J. Avigad and A. Chlipala, Eds. ACM Press, 2016, pp. 27–36.

[32] F. Haftmann and T. Nipkow, "Code generation via higher-order rewrite systems," in *FLOPS 2010*, ser. LNCS, M. Blume, N. Kobayashi, and G. Vidal, Eds., vol. 6009. Springer, 2010, pp. 103–117.

[33] M. Fleury and D. Kaufmann, "Isabelle pac formalization," theory files at https://bitbucket.org/isafol/isafol/src/master/PAC/, Accessed: 2020-05-06. [Online]. Available: http://people.mpi-inf.mpg.de/~mfleury/IsaFoL/current/PAC_Checker/PAC_Checker/index.html

[34] J. C. Blanchette, S. Böhme, M. Fleury, S. J. Smolka, and A. Steckermeier, "Semi-intelligible Isar proofs from machine-generated proofs," *J. Autom. Reasoning*, vol. 56, no. 2, pp. 155–200, 2016.

[35] P. Lammich, "Refinement to Imperative/HOL," in *ITP 2015*, ser. LNCS, C. Urban and X. Zhang, Eds., vol. 9236. Springer, 2015, pp. 253–269.

[36] S. Weeks, "Whole-program compilation in MLton," in *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*. ACM Press, 2006, p. 1.

[37] P. Lammich, "Generating verified LLVM from Isabelle/HOL," in *ITP 2019*, A. Tolmach, J. Harrison, and J. O'Leary, Eds., 2019.

[38] N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi, "Formal design of arithmetic circuits based on arithmetic description language," *IEICE Transactions*, vol. 89-A, no. 12, pp. 3500–3509, 2006.

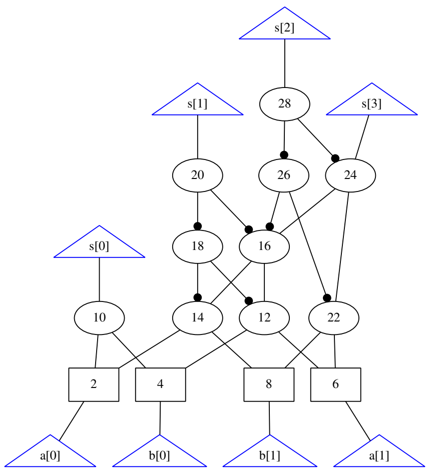
[39] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "BTOR2 , BtorMC and Boolector 3.0," in *CAV*, ser. LNCS, vol. 10981. Springer, 2018, pp. 587–595.

[40] A. Mahzoon, D. Große, and R. Drechsler, "Multiplier generator GenMul," http://www.sca-verification.org/, 2019.

[41] P. Lammich, "The GRAT tool chain - efficient (UN)SAT certificate checking with formal correctness guarantees," in *SAT*, ser. LNCS, vol. 10491. Springer, 2017, pp. 457–463.

[42] M. J. H. Heule, W. A. Hunt, Jr., M. Kaufmann, and N. Wetzler, "Efficient, verified checking of propositional proofs," in *ITP*, ser. LNCS, vol. 10499. Springer, 2017, pp. 269–284.

[43] A. Rebola-Pardo and J. Altmanninger, "Frying the egg, roasting the chicken: Unit deletions in DRAT proofs," in *CPP*, J. Blanchette and C. Hritcu, Eds. ACM, 2020.



Index	Polynomial equation	Gate constraint
1	$-s_3 + l_{24}$	$s_3 = l_{24}$
2	$-s_2 + l_{28}$	$s_2 = l_{28}$
3	$-l_{28} + l_{26}l_{24} - l_{26} - l_{24} + 1$	$l_{28} = \neg l_{26} \wedge \neg l_{24}$
4	$-l_{26} + l_{22}l_{16} - l_{22} - l_{16} + 1$	$l_{26} = \neg l_{22} \wedge \neg l_{16}$
5	$-l_{24} + l_{22}l_{16}$	$l_{24} = l_{22} \wedge l_{16}$
6	$-l_{22} + b_1a_1$	$l_{22} = b_1 \wedge a_1$
7	$-s_1 + l_{20}$	$s_1 = l_{20}$
8	$-l_{20} + l_{18}l_{16} - l_{18} - l_{16} + 1$	$l_{20} = \neg l_{18} \wedge \neg l_{16}$
9	$-l_{18} + l_{14}l_{12} - l_{14} - l_{12} + 1$	$l_{18} = \neg l_{14} \wedge \neg l_{12}$
10	$-l_{16} + l_{14}l_{12}$	$l_{16} = l_{14} \wedge l_{12}$
11	$-l_{14} + b_1a_0$	$l_{14} = b_1 \wedge a_0$
12	$-l_{12} + b_0a_1$	$l_{12} = b_0 \wedge a_1$
13	$-s_0 + l_{10}$	$s_0 = l_{10}$
14	$-l_{10} + b_0a_0$	$l_{10} = b_0 \wedge a_0$

Figure 4. AIG of a simple 2 bit multiplier (left) with corresponding polynomial equations (right).

Appendix

We present an example on how to generate PAC proofs as a by-product of circuit verification.

Example 2. *In this example we present how PAC proofs are generated as a by-product of multiplier verification. Figure 4 shows an And-Inverter-Graph (AIG) of a simple 2-bit multiplier. Nodes in the AIG represent logical conjunction and markings on the edges represent negation. For each node we introduce a corresponding polynomial equation, such that the Boolean roots of the polynomial are the solutions of the gate constraints and vice versa. For example from* $l_{26} = \neg l_{16} \wedge \neg l_{22}$ *we derive the polynomial equation* $-l_{26} + l_{16}l_{22} - l_{16} - l_{22} + 1 = 0$. *These polynomials are shown on the right side of Fig. 4 and define the initial constraint set.*

The multiplier is correct if we derive that the gate constraints imply the specification $-8s_3 - 4s_2 - 2s_1 - s_0 + 4a_1b_1 + 2a_0b_1 + 2a_0b_1 + a_0b_0 = 0$. *We apply backward rewriting of the specification to check whether the multiplier is correct. We refer to [9] for more details on the reduction process.*

The corresponding rewriting steps can be seen in Fig. 5, where we show how the corresponding PAC proof is generated on the fly. The left column "Remainder" shows the current remainder during multiplier verification. Initially the remainder is set to the specification polynomial of the multiplier. The second column depicts the index of the gate constraint that is used in the current reduction step. For example in the first row we reduce the specification by the polynomial with index 1 i.e., using multivariate polynomial division with remainder we calculate $8s_3+4s_2+2s_1+s_0-4a_1b_1-2a_0b_1-2a_0b_1-a_0b_0-8(-s_3+l_{24}) = -4s_2 - 8l_{24} - 2s_1 - s_0 + 4b_1a_1 + 2b_1a_0 + 2b_0a_1 + b_0a_0$. *We generate a PAC rule for the subtrahend* $-8(-s_3 + l_{24})$ *and the difference becomes the new remainder. We continue polynomial reduction until completion and check whether the final remainder is equal to zero. Subsequently we generate PAC rules to add up the conclusion polynomials of the multiplication rules that were generated for the subtrahends. The result of the last proof rule matches the circuit specification.*

Remainder	Red.	PAC rules
$-8s_3 - 4s_2 - 2s_1 - s_0 + 4a_1b_1 + 2a_0b_1 + 2a_0b_1 + a_0b_0$	1	$15 * 1, 8, -8s_3 + 8l_{24};$
$-4s_2 - 8l_{24} - 2s_1 - s_0 + 4b_1a_1 + 2b_1a_0 + 2b_0a_1 + b_0a_0$	2	$16 * 2, 4, -4s_2 + 4l_{28};$
$-4l_{28} - 8l_{24} - 2s_1 - s_0 + 4b_1a_1 + 2b_1a_0 + 2b_0a_1 + b_0a_0$	3	$17 * 3, 4, -4l_{28} + 4l_{26}l_{24} - 4l_{26} - 4l_{24} + 4;$
$-4l_{26}l_{24} + 4l_{26} - 4l_{24} - 2s_1 - s_0 + 4b_1a_1 + 2b_1a_0 + 2b_0a_1 + b_0a_0 - 4$	4	$18 * 4, 4l_{24} - 4, -4l_{26}l_{24} + 4l_{26} + 4l_{24}l_{22}l_{16} - 4l_{24}l_{22} - 4l_{24}l_{16} + 4l_{24} - 4l_{22}l_{16} + 4l_{22} + 4l_{16} - 4;$
$-4l_{24}l_{22}l_{16} + 4l_{24}l_{22} + 4l_{24}l_{16} - 8l_{24} + 4l_{22}l_{16} - 4l_{22} - 2s_1 - 4l_{16} - s_0 + 4b_1a_1 + 2b_1a_0 + 2b_0a_1 + b_0a_0$	5	$19 * 5, 4l_{22}l_{16} - 4l_{22} - 4l_{16} + 8, -4l_{24}l_{22}l_{16} + 4l_{24}l_{22} + 4l_{24}l_{16} - 8l_{24} + 4l_{22}l_{16};$
$-4l_{22} - 2s_1 - 4l_{16} - s_0 + 4b_1a_1 + 2b_1a_0 + 2b_0a_1 + b_0a_0$	6	$20 * 6, 4, -4l_{22} + 4b_1a_1;$
$-2s_1 - 4l_{16} - s_0 + 2b_1a_0 + 2b_0a_1 + b_0a_0$	7	$21 * 7, 2, -2s_1 + 2l_{20};$
$-2l_{20} - 4l_{16} - s_0 + 2b_1a_0 + 2b_0a_1 + b_0a_0$	8	$22 * 8, 2, -2l_{20} + 2l_{18}l_{16} - 2l_{18} - 2l_{16} + 2;$
$-2l_{18}l_{16} + 2l_{18} - 2l_{16} - s_0 + 2b_1a_0 + 2b_0a_1 + b_0a_0 - 2$	9	$23 * 9, 2l_{16} - 2, -2l_{18}l_{16} + 2l_{18} + 2l_{16}l_{14}l_{12} - 2l_{16}l_{14} - 2l_{16}l_{12} + 2l_{16} - 2l_{14}l_{12} + 2l_{14} + 2l_{12} - 2;$
$-2l_{16}l_{14}l_{12} + 2l_{16}l_{14} + 2l_{16}l_{12} - 4l_{16} + 2l_{14}l_{12} - 2l_{14} - 2l_{12} - s_0 + 2b_1a_0 + 2b_0a_1 + b_0a_0$	10	$24 * 10, 2l_{14}l_{12} - 2l_{14} - 2l_{12} + 4, -2l_{16}l_{14}l_{12} + 2l_{16}l_{14} + 2l_{16}l_{12} - 4l_{16} + 2l_{14}l_{12};$
$-2l_{14} - 2l_{12} - s_0 + 2b_1a_0 + 2b_0a_1 + b_0a_0$	11	$25 * 11, 2, -2l_{14} + 2b_1a_0;$
$-2l_{12} - s_0 + 2b_0a_1 + b_0a_0$	12	$26 * 12, 2, -2l_{12} + 2b_0a_1;$
$-s_0 + b_0a_0$	13	$27 * 13, -1, s_0 - l_{10};$
$-l_{10} + b_0a_0$	14	$28 * 14, -1, l_{10} - b_0a_0;$
0		$29 + 16, 17, -4s_2 + 4l_{26}l_{24} - 4l_{26} - 4l_{24} + 4;$
		$30 + 18, 19, -4l_{26}l_{24} + 4l_{26} - 4l_{24} + 4l_{22} + 4l_{16} - 4;$
		$31 + 29, 30, -4s_2 - 8l_{24} + 4l_{22} + 4l_{16};$
		$32 + 32, 22, -4s_2 - 8l_{24} + 4l_{16} + 4b_1a_1;$
		$33 + 21, 22, -2s_1 + 2l_{18}l_{16} - 2l_{18} - 2l_{16} + 2;$
		$34 + 23, 24, -2l_{18}l_{16} + 2l_{18} - 2l_{16} + 2l_{14} + 2l_{12} - 2;$
		$35 + 25, 26, -2l_{14} - 2l_{12} + 2b_1a_0 + 2b_0a_1;$
		$36 + 33, 34, -2s_1 - 4l_{16} + 2l_{14} + 2l_{12};$
		$37 + 36, 35, -2s_1 - 4l_{16} + 2b_1a_0 + 2b_0a_1;$
		$38 + 27, 28, -s_0 + b_0a_0;$
		$39 + 15, 32, -8s_3 - 4s_2 + 4l_{16} + 4b_1a_1;$
		$40 + 37, 38, -2s_1 - 4l_{16} - s_0 + 2b_1a_0 + 2b_0a_1 + b_0a_0;$
		$41 + 39, 40, -8s_3 - 4s_2 - 2s_1 - s_0 + 4b_1a_1 + 2b_1a_0 + 2b_0a_1 + b_0a_0;$

Figure 5. Generating PAC rules during multiplier verification.