SAT, Computer Algebra, Multipliers *

Daniela Kaufmann[1], Armin Biere[1], and Manuel Kauers[1]

Johannes Kepler University, Linz, Austria
daniela.kaufmann@jku.at armin.biere@jku.at manuel.kauers@jku.at

Abstract

Verifying multiplier circuits is an important problem which in practice still requires substantial manual effort. The currently most effective approach uses polynomial reasoning. However parts of a multiplier, i.e., complex final stage adders are hard to verify using computer algebra. In our approach we combine SAT and computer algebra to substantially improve automated verification of integer multipliers. In this paper we focus on the implementation details of our new dedicated reduction engine, which not only allows fully automated adder substitution, but also employs polynomial reduction efficiently. Our tool is furthermore able to generate proof certificates in the practical algebraic calculus and we also investigate the size of these proofs for one specific multiplier architecture.

1 Introduction

Formal verification of arithmetic circuits is extremely important to help to prevent issues like the famous Pentium FDIV bug. There have been many attempts since then to verify such circuits, but even today the problem of formally verifying arithmetic circuits, and especially multiplier circuits, is still considered to be hard and cannot be applied fully automated. In principle, theorem provers in combination with SAT are able to certify industrial multipliers [12]. However, such approaches lack automation.

Currently the most successful automated approach uses polynomial reasoning [4, 13, 17, 18, 23] and in recent years has seen significant progress. The approach of [17, 18] employs local cancellation of vanishing monomials in converging cones, which allows to verify a large variety of multiplier architectures much more efficiently than previous work. The authors of [4, 23] eliminate redundant polynomials by identifying full- and half-adders in the multipliers. This technique is able to verify large simple multipliers, but fails on even slightly more complex multiplier architectures.

In our method [14] we combine two approaches, i.e., SAT and computer algebra. We observe that final stage adders of multipliers are a real challenge for the algebraic approach as some adder designs rely on sequences of OR-gates, which lead to an explosion of the polynomial representation of the intermediate results. Contrarily SAT solvers can easily verify the equivalence of adder circuits. Therefore we apply adder substitution and replace complex final stage adders by simpler adders and verify the correctness of the substitution using SAT solvers. The correctness of the simplified multiplier is shown using computer algebra. Our method is an order of magnitude faster than related work and is able to verify circuits with input bitwidth 2048.

Our reduction engine AMULET [14] detects complex final stage adders and applies adder substitution fully automatically. A bit-level miter in conjunctive normal form (CNF) as well as a rewritten multiplier is generated. In the verification phase AMULET uses the structure of the polynomial representation of circuits and thus is more efficient in circuit verification than computer algebra systems [7, 22] used in our previous work. Additionally we apply preprocessing based on variable elimination.

Furthermore AMULET efficiently produces certificates in the PAC format [21], which allow to check the correctness of the verification results. None of the related work [4, 17, 18, 23] produces certificates.

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

This paper provides supplementary material for an invited talk at Vampire'19 of the second author based on [13, 14]. We discuss the implementation of AMULET and present the underlying algorithms of [14] in more detail. Additionally we provide a generalization of our incremental approach of [13]. We further show that we are able to generate proof certificates of quadratic length for simple multipliers.

2 Algebraic approach

We consider acyclic gate-level circuits C which implement integer multiplication. The circuits have $2n$ input bits $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1} \in \{0, 1\}$, as well as $2n$ output bits $s_0, \ldots, s_{2n-1} \in \{0, 1\}$ and further a number of internal logical gates denoted by $g_0, \ldots, g_k \in \{0, 1\}$. Let R be a commutative ring with unity and let $R[X]$ be the polynomial ring over R and the set of variables

$$X = \{ a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, s_0, \ldots, s_{2n-1}, g_0, \ldots, g_k \}.$$

A *term* $\tau = x_1^{d_1} \cdots x_r^{d_r}$ is a product of powers of variables for certain $d_1, \ldots, d_r \in \mathbb{N}$. A *monomial* is a multiple of a term $c\tau$, with $c \in R$ and a *polynomial* p is a finite sum of monomials.

An order $\leq$ is fixed on the set of terms such that $1 \leq \tau$ and $\sigma_1 \leq \sigma_2 \Rightarrow \tau\sigma_1 \leq \tau\sigma_2$ for all terms τ, σ_1, σ_2. Such an order is a *lexicographic term order* if for all terms $\sigma_1 = x_1^{d_1} \cdots x_r^{d_r}$, $\sigma_2 = x_1^{e_1} \cdots x_r^{e_r}$ it holds that $\sigma_1 < \sigma_2$ iff there exists i with $d_j = e_j$ for all $j < i$, and $d_i < e_i$. The largest term (w.r.t. $\leq$) in a polynomial $p = c\tau + \cdots$ is called the *leading term* $\mathrm{lt}(p) = \tau$. The *leading coefficient* and *leading monomial* of p are defined accordingly. Furthermore we call $p - c\tau$ the *tail of* p.

The *specification* of a circuit describes the desired relation between the outputs and inputs of a circuit. If for all possible inputs the circuit computes the desired output, we say that the circuit fulfills its specification and thus is correct. Formal verification aims to derive whether a circuit fulfills its specification or not. In the algebraic verification approach we model each logical gate by a polynomial. Correctness of the circuit is shown by deriving that the specification, also encoded as a polynomial $\mathcal{L}$, is implied by the gate polynomials.

The polynomial ring R is fixed with the specification. Although we model integer multiplication, we showed in [14] that it is beneficial to use more general polynomial rings which allow modular reasoning.

Definition 1. The specification $\mathcal{U}_n$ of n-bit unsigned integer multipliers in the ring $\mathbb{Z}_{2^{2n}}[X]$ is given as

$$\mathcal{U}_n = \sum_{i=0}^{2n-1} 2^i s_i - \left(\sum_{i=0}^{n-1} 2^i a_i\right)\left(\sum_{i=0}^{n-1} 2^i b_i\right) \tag{1}$$

As discussed in [14] modular reasoning also allows to define the specification of truncated multipliers, i.e., a truncated multiplier only returns the n least significant output bits.

Definition 2. The specification $\mathcal{T}_n$ of n-bit truncated multipliers in the ring $\mathbb{Z}_{2^n}[X]$ is given as

$$\mathcal{T}_n = \sum_{i=0}^{2n-1} 2^i s_i - \left(\sum_{i=0}^{n-1} 2^i a_i\right)\left(\sum_{i=0}^{n-1} 2^i b_i\right) = \sum_{i=0}^{n-1} 2^i s_i - \sum_{i=0}^{n-1} \sum_{j=0}^{n-1-i} 2^{i+j} a_i b_j. \tag{2}$$

After fixing the specification and thus the coefficient ring R, each logical gate in the circuit is encoded as a polynomial, such as:

$$\begin{array}{llll} u = \neg v & \text{implies} & 0 = -u + 1 - v & u = v \vee w \quad \text{implies} \quad 0 = -u + v + w - vw \\ u = v \wedge w & \text{implies} & 0 = -u + vw & u = v \oplus w \quad \text{implies} \quad 0 = -u + v + w - 2vw. \end{array} \tag{3}$$

The polynomial equations in (3) are chosen in such a way that the possible solutions with $u, v, w \in \{0, 1\}$ of the polynomials in $R[X]$ are the solutions of the gate constraints and vice versa. Note, the polynomials above are defined in the ring $\mathbb{Z}[X]$ and thus the structure may differ for different coefficient rings R. We order the terms according to a *reverse topological lexicographic term order*, such that the output variable of a gate is always greater than the variables attached to the input edges of that gate.

Definition 3. By $G(C) \subseteq R[X]$ we denote the set of *circuit polynomials* which contains for each gate of C the corresponding polynomial of (3). We further have *Boolean value constraints* $x(1-x) = 0$ for $x \in X$, encoding that x is a Boolean variable. Let $B(Y) = \{y(1-y) \mid y \in Y\} \subseteq R[X]$ for $Y \subseteq X$, be the set of Boolean value constraints for Y.

Definition 4. A nonempty subset $I \subseteq R[X]$ is called an *ideal* if $\forall\, p, q \in I : p + q \in I$ and $\forall\, p \in R[X] \,\forall\, q \in I : pq \in I$. A set $P = \{p_1, \ldots, p_s\} \subseteq R[X]$ is called a *basis* of I if $I = \{p_1q_1 + \cdots + p_sq_s \mid q_1, \ldots, q_s \in R[X]\}$. We say I is generated by P and write $I = \langle P \rangle$. The sum of two ideals I and J is defined as $I + J = \{p + q \mid p \in I, q \in J\}$.

In [14] we showed that the question whether $\mathcal{L}$ is implied by the gate polynomials of C and the Boolean value constraints can be answered by deciding a so-called ideal membership problem: "Given $q \in R[X]$ and a (finite) set of polynomials $P \subseteq R[X]$, decide whether $q \in \langle P \rangle$."

Definition 5. Let $P \subseteq R[X]$. If for a certain term order, all leading terms of P only consist of a single variable with exponent 1 and are unique and further all leading coefficients are multiplicatively invertible in R, then we say P has *unique monic leading terms* (UMLT). Let $X_0(P) \subseteq X$ be the set of all variables that do not occur as leading terms in P. We further define $B_0(P) = B(X_0(P))$.

It is easy to see that for an acyclic circuit C the set $G(C)$ has UMLT for a fixed reverse topological term order. Further $X_0(P)$ contains only circuit inputs a_i, b_i.

Definition 6. Let C be a circuit and let $J(C) = \langle G(C) \cup B_0(C) \rangle \subseteq R[X]$, with $B_0(C) = B_0(G(C))$.

Corollary 1. [14] A circuit C fulfills $\mathcal{L}$ iff $\mathcal{L} \in J(C)$.

The theory of Gröbner bases [3] offers a decision procedure for the ideal membership problem. For our purpose we use the more general theory of D-Gröbner bases [2], where the coefficient domain D is a principal ideal domain (PID). Let $p, q, r \in D[X]$ and let $P \subseteq D[X]$. A basis P of an ideal $I \subseteq D[X]$ is a *D-Gröbner basis* of I iff $\forall q \in I\ \exists p \in P : \mathrm{lm}(p) \mid \mathrm{lm}(q)$. Every ideal of $D[X]$ has a D-Gröbner basis, and there is an algorithm (Thm. 10.14 of [2]) which, given an arbitrary basis of an ideal, computes a D-Gröbner basis of it in finitely many steps.

We say q *D-reduces* to r w.r.t. p if there exists a monomial m' in q with $m' = m\,\mathrm{lm}(p)$ and $r = q - mp$. The remainder r of the D-reduction of q by P is such that $q - r \in \langle P \rangle$ and r is *D-reduced* w.r.t. P. If P is a D-Gröbner basis, then $r = 0$ iff $q \in \langle P \rangle$.

For the specifications listed in Def. 1 and Def. 2 we fixed the polynomial rings to $\mathbb{Z}_l[X]$ for $l \in \mathbb{N}$. In general $\mathbb{Z}_l$ is not a PID, but we showed in [14] that the ideal membership problem in $\mathbb{Z}_l[X]$ can be converted to an ideal membership problem in the ring $\mathbb{Z}[X]$, with $\mathbb{Z}$ being a PID. Whenever we want to decide whether a polynomial $q \in I \subseteq \mathbb{Z}_l[X]$ we can instead check whether $q \in I + \langle l \rangle \subseteq \mathbb{Z}[X]$. For the latter we have the concept of D-Gröbner bases available. And since $G(C)$ has UMLT we can directly derive a D-Gröbner basis for $J(C) + \langle l \rangle$.

Lemma 1. [14] Let $l \in \mathbb{N}$. Then $G(C) \cup B_0(C) \cup \{l\}$ is a D-Gröbner basis for $J(C) + \langle l \rangle \subseteq \mathbb{Z}[X]$.

2.1 Incremental Verification

In [13] we introduced an incremental verification algorithm, which splits the verification problem into smaller more manageable subproblems by partitioning the circuit into column-wise slices and splitting the word-level specification into multiple smaller specifications which relate the partial products, incoming carries, sum output bit and the outgoing carries of each slice. However this algorithm is tailored to multiplication of unsigned bit-vectors. In this section we show how to apply this procedure to different multiplier specifications. As the number of output bits varies for different multipliers, eg. in Def. 2, we denote the number of output bits by the constant m and fix $l = 2^m$ in this section.

Definition 7. Let $I_i := \{\text{gate } g \mid g \text{ is in input cone of } s_i\}$ be the input cone of each output bit s_i for $0 \leq i < m$. A slice S_i is defined as the difference of consecutive cones I_i, i.e., $S_0 := I_0$ and $S_{i+1} := I_{i+1} \setminus \bigcup_{j=0}^{i} S_j$.

Definition 8 (Sliced Gröbner Bases)**.** Let $G_i(C)$ be the set of circuit polynomials of the gates in a slice S_i. The terms are ordered such that the requirements of Lemma. 1 are fulfilled. We define by $X_0(G_i)$ the set of variables that do not occur as leading terms in $G_i(C)$ and further define $B_0(G_i) = B(X_0(G_i))$.

Corollary 2. $G_i(C) \cup B_0(G_i) \cup \{2^m\}$ is a D-Gröbner basis for $\langle G_i(C) \cup B_0(G_i)\rangle + \langle 2^m\rangle$.

Corollary 2 follows directly from Lemma. 1. It is easy to see that $\langle G_i(C) \cup B_0(G_i)\rangle$ contains all the Boolean value constraints $B(G_i)$ for the gate variables in S_i, thus we may use them in the reduction process to eliminate exponents greater than 1 in the intermediate reduction results. After splitting the circuit, we are now going to split the word-level specification of a multiplier.

Definition 9. Let C be a multiplier circuit which is sliced according to Def. 7 and let $\mathcal{L}$ be the specification of C. For slice S_i with $0 \leq i < m$ let $P_i = \sum_{j+k=i} \alpha_{jk} a_j b_k$ be the *partial product sum* of column i, where the constant α_{jk} is the coefficient of the term $a_j b_k$ in $\mathcal{L}$.

Definition 10. Let C be a multiplier circuit. A sequence of $m+1$ polynomials $C_0, \ldots, C_m$ over the variables of C is called a *carry sequence* if for all $0 \leq i \leq m$ it holds that

$$-C_i + C_{i+1} + \alpha_i s_i + P_i \quad \in J(C)$$

where the constant α_i is the coefficient of s_i in $\mathcal{L}$. We call the polynomials $-C_i + C_{i+1} + \alpha_i s_i + P_i$ the *carry recurrence relations* for the sequence $C_0, \ldots, C_m$.

It remains to fix the boundary polynomial C_m, where we simply choose $C_m = 0$. Our incremental algorithm is shown in Alg. 1 and it follows from the proof of Thm.6 in [13] that Alg. 1 is correct.

Algorithm 1: Multiplier Checking Algorithm

Input : Circuit C with m output bits, sliced Gröbner bases G_i
Output: Determine whether C is a correct multiplier

$l \leftarrow 2^m$, $C_m \leftarrow 0$;
for $i \leftarrow m-1$ **to** 0 **do**
  $C_i \leftarrow \text{Remainder}(C_{i+1} + \alpha_i 2^i s_i - 2^i P_i,\ G_i(C) \cup B(G_i) \cup \{l\})$
return $C_0 = 0$

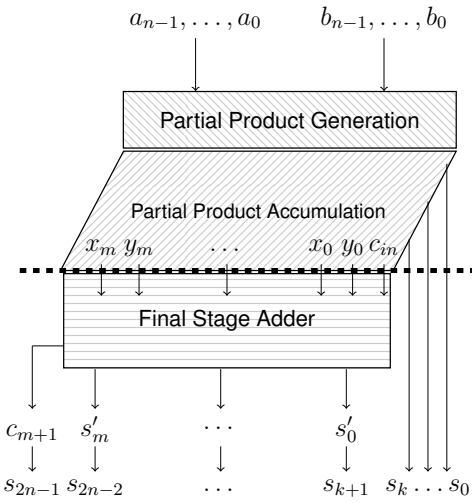


Figure 1: The components of a multiplier.

3 SAT

Computer algebra is able to verify simple multipliers very efficiently. However more complex multiplier architectures still impose quite a challenge and lead to a monomial blow-up in the intermediate reduction results. The reason for this blow-up are certain adder structures, which are part of the multipliers. During preparation for the SAT Race 2019 [15] we observed that checking the equivalence of different adder circuits is rather trivial for SAT solvers. We make use of this observation in the verification procedure and combine computer algebra and SAT. We summarize the main idea, as presented in [14].

Generally multipliers can be decomposed into three components [20], which are shown in Fig. 1. In the first component *partial product generation* (PPG) the partial products $a_i b_j$ as contained in $\mathcal{L}$ are derived. This can for example be achieved using simple AND-gates or using a more complex Booth encoding. In the second stage *partial product accumulation* (PPA) the partial products are reduced to two layers using full- and half-adders. In the last stage the output of the circuit is computed using an adder circuit. Hence we call this component *final stage adder* (FSA).

Adder circuits can be split into two groups. Either the carries are computed simultaneously with the sum bits or they are calculated separately before the sum to decrease the latency of carry computation. A scheme for both adder types can be seen in Fig. 2. Adders of the first group are usually based on a sequence of half- and full-adders, which gives them a simple but inefficient structure, eg., ripple-carry adders. Adders of the second group are also called generate-and-propagate (GP) adders. In a GP adder with inputs $x_0, \ldots, x_m, y_0, \ldots, y_m, c_{in}$ and outputs $s'_0, \ldots, s'_m, c_{out}$ the output bits s'_i are calculated as $s'_i = p_i \oplus c_i$, with $p_i = x_i \oplus y_i$. The carries c_i are recursively generated using the equation $c_i = (x_{i-1} \wedge y_{i-1}) \vee (c_{i-1} \wedge p_{i-1})$ with $c_{m+1} = c_{out}$ and $c_0 = c_{in}$. The precise derivation of the carries c_i (recursively, unrolled or mixed) depends on the architecture of the adders, but is generally computed using sequences of OR-gates. These sequences of OR-gates make the GP adders hard to verify using the algebraic approach as the following example shows.

Example 1. Let $o = o_2 \vee x_0, o_2 = o_1 \vee x_1, o_1 = x_3 \vee x_2$ represent a sequence of three OR-gates, which can be simplified to $o = x_0 \vee x_1 \vee x_2 \vee x_3$. The corresponding polynomial representation $o = x_0 + x_1 - x_0x_1 + x_2 - x_0x_2 - x_1x_2 + x_0x_1x_2 + x_3 - x_0x_3 - x_1x_3 + x_0x_1x_3 - x_2x_3 + x_0x_2x_3 + x_1x_2x_3 - x_0x_1x_2x_3$ contains $2^4 - 1$ monomials.

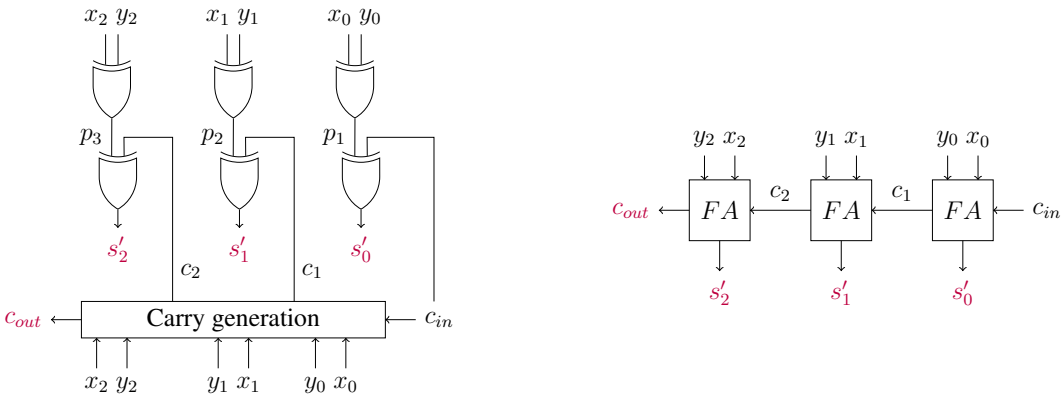


Figure 2: GP adder (left) and equivalent RCA (right).

In our approach we identify whether the FSA is a GP adder, using the equations $s'_i = p_i \oplus c_i$ and $p_i = x_i \oplus y_i$. The algorithm is described in detail in Sect. 4, where we present our tool AMULET. If we detect that the FSA is a GP adder, we substitute the FSA by a simple ripple-carry adder (RCA), which has the same inputs $x_0, \ldots, x_m, y_0, \ldots, y_m, c_{in}$ than the original FSA. We do not change the first two stages PPG and PPA. To prove that the RCA is equivalent to the GP adder we generate a bit-level miter in CNF, which is verified by a SAT solver. However, if the FSA is not a GP adder we do not apply adder substitution. After substitution we verify the rewritten AIG in AMULET using computer algebra. Figure 3 shows the original multiplier (orange) as well as the RCA and the bit-level miter (green). The dashed boxes depict which components of the extended multiplier are verified using SAT (blue) and computer algebra (red).

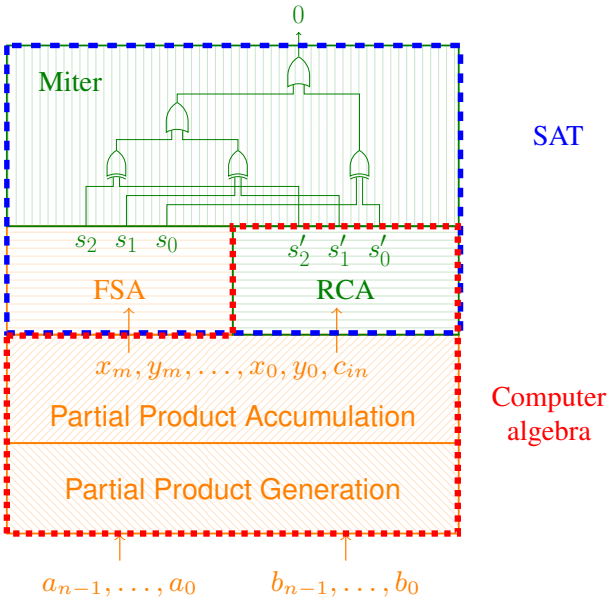


Figure 3: Reasoning techniques used to verify the extended multiplier circuit.

Algorithm 2: Adder substitution AMULET

Input : Circuit C in AIG format
Output: Rewritten Circuit C' in AIG format, bit-level miter as CNF F

1 Init(C);
2 $\tau \leftarrow 1$;
3 $c_{out}, \tau \leftarrow$Identify-Carry-Out(s_{2n-1});
4 **if** $\tau = 0$ **then**
5 **return** $C, 0$;
6 $j \leftarrow 2n-2, \sigma \leftarrow 1$;
7 **while** σ **and** $j \geq 0$ **do**
8 $\sigma \leftarrow$ Check-if-XOR (s_j);
9 $\sigma, c_j, p_j \leftarrow$ Identify-p$_j$-and-c$_j$ (s_j, σ);
10 $\sigma, x_j, y_j \leftarrow$Mark-Adder-Inputs (p_j, σ);
11 $j \leftarrow j - 1$;
12 $c_{in} \leftarrow c_j$;
13 $\tau \leftarrow$ Follow-and-Mark-Paths(c_{out}, X, Y, c_{in});
14 **for** $i \leftarrow j+1$ **to** $2n-2$ **do**
15 $\tau \leftarrow$ Follow-and-Mark-Paths(s_i, X, Y, c_{in}, τ);
16 **if** $\tau = 0$ **then**
17 **return** $C, 0$;
18 $R \leftarrow$ Generate-AIG-RCA(X, Y, c_{in});
19 $M \leftarrow$ Generate-Miter(C, R);
20 $F \leftarrow$ Miter-to-CNF(M);
21 $C' \leftarrow$ Generate-Rewritten-AIG(C, R);
22 **return** C', F

4 AMulet

In this section we explain implementation details of our tool AMULET. Our tool, which is written in C reads multipliers given as And-Inverter-Graphs (AIG) [16] and automatically applies adder substitution and verification. Additionally we are able to generate proof certificates.

4.1 Adder Substitution

Our algorithm, which identifies whether the FSA is a GP adder and, if necessary, replaces the FSA by a RCA is shown in Alg. 2. It reads the original multiplier and returns a circuit in the AIG format as well as a CNF. To identify GP adders we highly relate on their structure as presented in Sect. 3. In particular we rely on the fact that the outputs s'_i are always outputs of XOR-gates and that the carries c_i are never outputs of XOR-gates.

In the initialization phase AMULET reads the given multiplier and for each node in the AIG we introduce a unique variable. Variables in AMULET are organized in an ordered array, where the indices match the literal (divided by 2) of the AIG node. As there is a one-to-one correspondence between variables and AIG nodes we will use both terms interchangeably. We further identify whether the variable is an output or an internal gate of an XOR gate, using syntactic pattern matching.

The variable τ of line 2 acts as an error-flag. In line 3 we identify whether the output s_{2n-1} of the multiplier is the carry output c_{out} of the FSA, which is not always the case. In some multiplier architectures the output s_{2n-1} is computed as an XOR, whose inputs are the carry output c_{out} of the FSA and some output from the PPA step, which is usually again an XOR gate. Thus in line 3 we identify whether s_{2n-1} is an XOR gate. If not, then $s_{2n-1} = c_{out}$. If on the other hand s_{2n-1} is an XOR gate we

examine the inputs of s_{2n-1} and identify which child is not an XOR gate. This child is then identified as c_{out}. If both inputs are not XOR gates, we cannot clearly identify c_{out} of the FSA and set $\tau = 0$. In that case the algorithm terminates and returns the original multiplier and an empty bit-level miter.

In the while-loop we identify the inputs $x_0, \ldots, x_m, y_0, \ldots, y_m, c_{in}$ of the FSA. We do not know the concrete value of m in advance, as it depends on the multiplier architecture. Hence we recursively iterate over the outputs of the multiplier. We start the loop at the output s_{2n-2}, since s_{2n-1} was used to identify the carry output of the FSA. In line 8 we check if the output s_j is an XOR gate. If so, we identify the propagate bit p_j and the carry bit c_j in the next step. Here we rely on the fact that p_j is an XOR gate and c_j is not an XOR gate. Using p_j we mark the inputs x_j, y_j of the adder in the next step.

As shown in Fig. 1 not all output bits of the multiplier are computed by the FSA. Smaller output bits may already be computed in the PPA step. Hence at some point we are not able to identify p_j, c_j or x_j, y_j anymore, which we capture in σ. If the FSA is not a GP adder the loop will directly stop after the first iteration. The carry-in c_{in} of the FSA is set to the smallest c_j, which was identified.

In lines 13 to 15 we mark all gates which belong to the FSA. We start at the carry output c_{out} resp. sum outputs $s_{j+1}, \ldots, s_{2n-1}$ and follow all paths in the input cones until we either reach a marked input x_i, y_i or c_{in}. We mark the visited variables. If at some point we reach the input variables a_i, b_i of the multiplier the FSA is not a GP adder, i.e., we were not able to clearly identify the boundaries of the FSA. Consequently adder substitution was not successful and the initially given AIG is returned without generating a bit-level miter. If on the other hand all paths stop at the marked inputs or at c_{in}, we have successfully identified and marked all gates belonging to a GP adder and apply adder substitution.

We generate an equivalent RCA in line 18. A RCA is simply a sequence of full-adders, cf. Fig. 2 and the AIG encoding of a full-adder can be seen in Fig. 5b. After the RCA is generated, the bit-level miter is defined. It contains all the gates which are identified to belong to the GP adder and the gates of the RCA. Furthermore we add XOR gates, whose inputs are corresponding pairs of output bits of the two adders. These XOR gates are summed up by a sequence of OR-gates, cf. Fig. 3. If the two adders are equivalent, all equivalent pairs of output bits compute the same result. Thus the outputs of the XOR gates are 0, consequently all OR-gates are 0 and the output of the miter is 0.

The equivalence of the adders is verified using a SAT-solver. Hence the bit-level miter is translated into a CNF F in line 20. More precisely, the propositional formulas represented by each AIG node are translated into CNF, which is rather straightforward. If for example an AIG node represents $x = a \wedge \overline{b}$, the equivalent propositional formula is $\neg(x \leftrightarrow a \wedge \overline{b}) = \top$ which can be translated to the CNF $(x \vee \overline{a} \vee b) \wedge (\overline{x} \vee a) \wedge (\overline{x} \vee \overline{b}) = \top$. We iterate over each node and output the corresponding clauses in DIMACS format. The final clause which is added, is the assumption that the output of the miter is 1. Thus for a correct adder substitution the SAT solver has to return that the CNF is UNSAT.

In the rewritten multiplier, we keep all nodes of the original multiplier, which are not marked to be an element of the FSA and replace the subgraph defining the GP adder by the AIG of the RCA. The rewritten AIG C' as well as the CNF F are returned by AMULET.

4.2 Verification

After applying adder substitution the multiplier is verified. The pseudo-code can be seen in Alg. 3. During initialization, which is similar to Alg. 2, we fix the specification polynomial $\mathcal{L} \in \mathbb{Z}_l[X]$ and thus the constant l. As there are now more data structures involved, in particular representation of polynomials, let us briefly discuss our design decisions. The variables are organized as an ordered array. Terms are represented as ordered linked lists of variables. In general terms will be used multiple times during the reduction process, thus they are organized in a hash table. Monomials contain a coefficient and a term. Since the values of the coefficients exceed 2^{64} we use the GMP library [8] for number representation. Polynomials are represented as sorted linked lists of monomials. In the data structure

Algorithm 3: Outline of verification flow in AMULET

```
   Input  : Substituted Circuit C in AIG format
   Output: Determine whether C is a multiplier
1  L, l ← Init(C);
2  for i ← 0 to 2n − 1 do
3  |  S_i ← Define-Slices(i);
4  |  Order-Slices (S_i);
5  |  G_i(C) ← Init-Polynomials-of-Slices (S_i);
6  Ω ← Search-for-Booth-Encoding (C);
7  for i ← 0 to 2n − 1 do
8  |  Local-Elimination (G_i(C), l);
9  Global-Elimination (Ω);
10 C_0 ← Incremental-Reduction (L, G_i(C));
11 return C_0 = 0
```

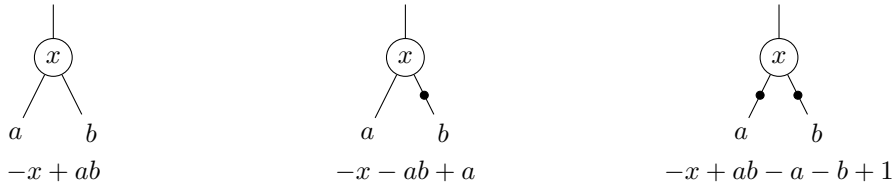


Figure 4: All polynomial encodings covered by AIG nodes

slice we store the gates which are assigned to a slice and the corresponding gate polynomials.

We use our incremental verification approach, cf. Alg. 1. Hence we define the slices as differences of consecutive input cones as introduced in Def. 7. However in certain cases not all gates are assigned to the correct slices. To this end we *merge* and *promote* gates as described in [13]. Additionally we identify which nodes are *carry* nodes, i.e., which nodes are used as inputs of nodes in bigger slices.

After the slices are defined we fix the reverse topological lexicographic term ordering in line 4. The gates inside the slices are ordered according to their reverse topological appearance and slices are ordered in descending order. Thus also the total order of the variables is reverse topological. As a consequence the polynomials $G_i(C)$, which are introduced in line 5, automatically form a D-Gröbner basis. Each AIG node represents an AND-gate between two inputs, which may or may not be inverted. Consequently three different polynomials are possible, as can be seen in Fig. 4. For each node we introduce the corresponding polynomial with x, a, b replaced by the corresponding variables. We further add for each output s_i a *linking polynomial* $-s_i + g_k$ to clearly mark which AIG node represents an output bit. All these polynomials mark our initial constraint set, i.e., the set $G(C)$ of Def. 3.

We apply syntactic pattern matching to identify whether the partial products are generated using a Booth encoding. Patterns which define Booth encoding usually stretch over more than one slice and we want to eliminate these nodes during "Global-Elimination" to reduce the size of the carries.

Before we apply "Global-Elimination" we locally eliminate variables in the sliced Gröbner bases $G_i(C)$. We described in [14] a procedure which allows us to locally eliminate variables without violating the D-Gröbner basis property.

Theorem 1 ([14])**.** Let $P \subseteq \mathbb{Z}[X]$ be a D-Gröbner basis of $\langle P \rangle$ with UMLT. Let $q \in P$ be a polynomial with $\mathrm{lt}(q) = z$ and no other polynomial $p \in P$ contains z. Then $P \setminus \{q\}$ is a D-Gröbner basis with UMLT for the ideal $J = I \cap \mathbb{Z}[X \setminus \{z\}]$.

Algorithm 4: Local-Elimination

Input : Ordered sliced Gröbner bases G_i, constant l
Output: Simplified ordered sliced Gröbner bases G_i

$p_0, \ldots, p_m \leftarrow$ Ordered-List-of-Polynomials$(G_i(C))$;
$\tau \leftarrow 1$;
while τ **do**
  $\tau \leftarrow 0$;
  for $j \leftarrow 0$ **to** m **do**
    if *Check-for-Elimination*(p_j) **then**
      $q \leftarrow$ Find-Parent-Polynomial$(p_0, \ldots, p_{j-1})$;
      $q \leftarrow$ D-reduction(q, p_j, l);
      $G_i(C) \leftarrow G_i(C) \setminus \{p_j\}$;
      $\tau \leftarrow 1$;
return $G_i(C)$

Algorithm 5: D-reduction in AMULET

Input : Two polynomials p and $q \in \mathbb{Z}[X]$, constant l
Output: Remainder r of D-reducing p modulo q

$p_d \leftarrow$ Divide-by-lm(p, q);
$p_m \leftarrow$ Multiply(p_d, q, l);
$r \leftarrow$ Add(p, p_m, l);
return r

We use the conclusion of Thm.1 as follows. Assume $z \in X$ shall be eliminated and let $p, q \in G_i(C)$ be such that $\mathrm{lt}(p) = z$ and z is contained in q. To eliminate z of $G_i(C)$, we D-reduce q by p and subsequently delete the polynomial p.

The pseudo-code for "Local-Elimination" is shown in Alg. 4. We iterate over the polynomials p_j in $G_i(C)$ and check whether the variable of the leading term is a candidate for local elimination, i.e. we check if the variable is contained in exactly one other polynomial of the same slice and if it is not marked as a carry variable. If both checks succeed, we search for the polynomial q, which contains the leading term of p_j and apply D-reduction of q by p_j. Since the polynomials are ordered, we only have to consider polynomials $p_i > p_j$ in this search.

Algorithm 5 shows how D-reduction is implemented in AMULET. In "Divide-by-lm" we use the UMLT property. Let $v = \mathrm{lt}(q)$. We iterate over the monomials m in p and check whether v is contained in m. If v is contained in m we generate a monomial m' which consists of all variables of m different from v. Furthermore we set $\mathrm{coeff}(m') = \mathrm{coeff}(m)$. All these generated monomials m' are summed up and define the polynomial p_d. The operations "Multiply" and "Add" correspond to the elementary polynomial operations. For addition we use the fact that polynomials are ordered lists of monomials. We iterate over the two polynomials simultaneously and merge them in an interleaved way. More precisely, we start at the leading monomials of p and p_m and compare them. If the monomials are different, we add the larger monomial to r. If the monomials are equal we generate a new monomial, which has the same term and the coefficient is the sum of the two coefficients. This way we ensure that r is again ordered. For multiplication we multiply each monomial of p_d with each monomial of q and sort the calculated monomials. In both operations we directly divide the calculated coefficients by the constant l in order to achieve reduction by l. We further handle reduction by $B(G_i)$ implicitly, i.e., we replace x^i by x during multiplication too, for all $i > 0$.

Example 2. Let $p = -a + 2bc - bd$ and $q = -b + 2xy \in \mathbb{Z}_4[X]$. The intermediate results of Alg. 5 are $p_d = 2c - d$, $p_m = -2bc + bd - dxy$ and $r = -a - dxy$.

Let us continue the discussion of Alg. 4. The polynomial q is replaced by the remainder of the D-reduction step and the polynomial p_j is eliminated from the sliced Gröbner basis $G_i(C)$. We repeat variable elimination until no more polynomial in $G_i(C)$ can be considered for local elimination, i.e., all variables of its leading term are either carries or contained in multiple polynomials of the same slice. The rewritten Gröbner basis $G_i(C)$ is returned.

Now consider Alg. 3, where we repeat "Local-Elimination" for all sliced Gröbner bases $G_i(C)$. In "Global-Elimination" we eliminate the variables which were previously marked, independently how often they occur of whether they are carries. To this end we have to iterate over all polynomials in $G(C)$, finding their parent polynomials for D-reduction.

After variable elimination we apply the incremental checking algorithm as presented in Alg. 1. We start with s_{2n-1} and apply D-reduction by the polynomials in $G_{2n-1}(C)$. In order to consider each polynomial of a slice only once for D-reduction, we D-reduce by the polynomials in $G_i(C)$ in reverse topological order. After we applied D-reduction by all polynomials of a slice we add to the remainder C_i the partial products and output bit of the next smaller slice in order to derive $C_i + \alpha_{i-1} 2^{i-1} s_{i-1} - 2^i P_{i-1}$. After reducing by $G_0(C)$, we check whether the final result is 0.

5 Proof Generation

Formal verification derives correctness of a given system. However, the process of verification as well as the implementation might not be bug-free. A common approach to increase the confidence in automated reasoning tools is to generate proof certificates, which are checked by independent proof checkers.

For example, providing certificates of unsatisfiability is mandatory in the SAT competition since 2013. Generating and checking proofs efficiently is a lively research topic in the SAT community and several proof formats such as RUP [9], DRUP [11], DRAT [10] and LRAT [6] are available.

In order to provide proof certificates for reasoning tools using computer algebra we developed in [21] a proof format, called practical algebraic calculus (PAC), which is based on the polynomial calculus [5] and captures whether a polynomial is contained in the ideal generated by a given set of polynomials.

Our tool AMULET is able to generate proof certificates in the PAC format [21] to validate the result of Alg. 3. These proofs can be checked by our independent proof checker PACTRIM [21]. We write proofs as sequences, where each rule is of the following form:

$d + : p_i, p_j, p_i + p_j;$ p_i, p_j appearing earlier in the proof or are contained in $G(C) \cup \{l\}$ and $p_i + p_j$ being reduced by $B(X)$

$d * : p_i, q, qp_i;$ p_i appearing earlier in the proof or is contained in $G(C) \cup \{l\}$ and $q \in R[X]$ being arbitrary and qp_i being reduced by $B(X)$

These rules model the properties of an ideal, as given in Def. 4. Thus every conclusion polynomial $p = p_i + p_j$ or $p = qp_i$ is an element of $\langle G(C) \cup \{l\} \rangle$. We extend the proof rules by an optional deletion information d, similar to clause deletion in [11]. If d occurs in a proof rule the antecedents p_i and p_j are deleted from the inference set, which helps to reduce the memory usage of PACTRIM.

We do not explicitly write down proof rules when reducing a Boolean value constraint. Similar to verification, reduction by $B(X)$ is computed implicitly, e.g., $* : x, x, x;$ is a valid proof rule.

Definition 11. The *length* of a PAC proof is defined as the number of generated proof rules. The *size* is determined as the total number of monomials in the conclusion polynomials, counted with repetition and *degree* defines the maximum degree seen in the conclusion polynomials.

PAC proofs are generated in AMULET as follows. The set of polynomials $G(C) \cup \{l\}$, which are defined in line 5 of Alg. 3 determines the initial constraints set. The specification $\mathcal{L}$ defines the target polynomial, i.e., the polynomial which is checked whether it is inferred by the proof rules. Proof rules have to be generated whenever polynomials are manipulated, that is for variable elimination, either locally or globally and during incremental reduction in Alg. 3.

For variable elimination we produce proof rules which simulate D-reduction of a polynomial p by a polynomial q, cf. Alg. 5. Note that p and q are both contained in $G(C)$ and thus appear earlier in the proof. In general two rules are generated, a multiplication rule and an addition rule:

$$(d) * : q, p_d, p_m; \qquad d + : p, p_m, r;$$

In AMULET reducing the polynomials p_m and r by the constant l is handled implicitly. However to generate a complete PAC proof, we need to generate explicit proof rules which model D-reduction of p_m and r by the constant l.

After a polynomial q was used for D-reduction during "Local-Elimination" we know, that we do not have to consider q anymore, as p was the only polynomial containing the leading variable of q. Thus we can delete q from the constraints set, which we indicate by the optional parameter "d". For "Global-Elimination" we have to be more careful with deletion, as the polynomial q may be used multiple times for elimination. In both cases we eliminate p as we want to continue with the rewritten polynomial r.

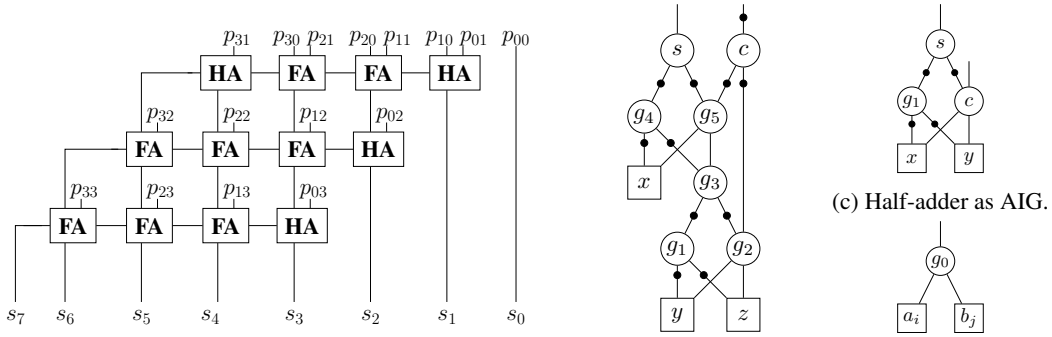
For monitoring the incremental reduction we also have to generate proof rules which simulate D-reduction of p by q. However in contrast to variable elimination, p is not part of the constraints set and thus the addition rule would raise an error. On the other hand recall that all elements of an ideal can be represented as a linear combination of the generators of the ideal, cf. Def 4. To simulate the linear combination we generate a multiplication PAC rule $(d) * : p_d, q, p_m$; for each D-reduction step and store the computed factor p_m. After finishing D-reduction of a slice S_i, we sum up all the generated factors p_m to derive the carry recurrence relations. After deriving all carry recurrence relations we sum them up and if the circuit is correct the final polynomial is the specification of the circuit. In both cases we sum up the polynomials in a tree-like approach, i.e., $((p_1 + p_2) + (p_3 + p_4))$ which is more beneficial compared to summing up the polynomials in order $(((p_1 + p_2) + p_3) + p_4)$ as this keeps the number of monomials in the intermediate summands smaller.

6 Proof Size

Proof complexity aims to analyze computational resources and allows to reason about the performance of solvers. In this section we want to elaborate the efficiency of AMULET and investigate the complexity of the generated proofs. In particular we are interested in the proof length, proof size and degree.

Proof complexity for multiplier circuits is for example studied in [1], where it is shown that verifying ring properties, e.g., commutativity of multiplication, admit polynomial resolution proofs for simple multipliers. Motivated by this result we experimentally show in [21] that checking commutativity of simple multipliers generates PAC proofs of quadratic length and cubic size. However these proofs are generated using existing computer algebra systems [22].

In this section we investigate the complexity of the proofs generated by AMULET for specific family of multipliers, more precisely btor-multipliers, which implement multiplication of unsigned integers. These multipliers are generated by Boolector [19] and have a simple architecture as can be seen in Fig. 5a for input bitwidth 4. They are also used in the experiments of [21] and correspond to the array multipliers as defined in [1]. In contrast to [1, 21] we investigate the complexity for verifying the correctness of the circuit. For the proof length and degree we can give a precise bound while for proof size we derive an upper bound.



(a) Outline of a 4-bit btor-multiplier (b) Full-adder as AIG. (c) Half-adder as AIG. (d) Partial Products.

Figure 5: The architecture of btor-multipliers and their representation as AIGs.

Lemma 2. Let C be a n-bit btor-multiplier. C contains n half-adders and $n^2 - 2n$ full-adders.

Proof. As Fig. 5a shows, we can clearly identify rows and columns in the btor-multipliers. Let Ab_i denote the sequence of all partial products $a_j b_i$ for $0 \leq j \leq n-1$. The first row of full- and half-adders (as seen from the circuit inputs) in C sum up the partial products Ab_0 and Ab_1. In row k for $k \geq 2$ the partial products Ab_k are added to the sum-outputs of the adders of row $k-1$. Thus a btor-multiplier consists of $n-1$ rows.

In row k with $k \geq 2$ the adders sum up two bitvectors of length n, which requires n adders. As we do not have an incoming carry the first adder is a half-adder and the remaining $n-1$ adders are full-adders. In the first row the partial product $a_0 b_0$ is directly processed to be output s_0. Thus only $2n-1$ bits are summed up, which requires 2 half-adders and $n-2$ full-adders. Consequently btor-multipliers have $(n-2)(n-1) + n - 2 = n^2 - 2n$ full-adder and $n - 2 + 2 = n$ half-adders. □

Lemma 3. Let C be a btor-multiplier of input bitwidth n. The number of variables is $8n^2 - 7n$ and the size of $G(C)$ is $8n^2 - 9n$.

Proof. The total number of variables consists of the input variables $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}$, output variables $s_0, \ldots, s_{2n-1}$ and the internal variables $g_0, \ldots, g_k$. It is easy to see that we need $4n$ variables for the inputs and outputs. The internal nodes either represent partial products or they represent internal nodes of full- and half-adders, cf. Fig. 5d,5b and 5c. Generating a partial product needs one variable $p_{ij} = a_i b_j$, thus n^2 variables are needed to identify the partial products. According to Lemma 2 btor-multipliers have n half-adders, each consisting of 3 nodes and $n^2 - 2n$ full-adders consisting of 7 nodes. Hence the total number of variables is $4n + n^2 + 3n + 7(n^2 - 2n) = 8n^2 - 7n$.

Each variable, despite of the $2n$ input variables, generates either a gate polynomial or a linking polynomial. Thus we have $8n^2 - 9n$ polynomials. □

For proof length we measure the number of generated PAC rules. Because of the specific structure of btor-multipliers D-reduction by the constant $l = 2^{2n}$ is not necessary. Thus each D-reduction step in variable elimination and in the incremental reduction algorithm produces at most two proof rules, namely one multiplication rule and one addition rule. Furthermore the partial products are generated using AND-gates, thus "Global-Elimination" is not necessary and proof rules are only generated in "Local-Elimination" and "Incremental-Reduction". Hence each gate constraint in $G(C)$ is considered only once for D-reduction and thus we have an upper bound of $2(8n^2 - 9n) = 16n^2 - 18n$ proof rules. This bound is not tight as the following lemma shows.

Theorem 2. The proof length of n-bit btor-multipliers produced in AMULET is $16n^2 - 20n - 1$.

Proof. In "Local-Elimination" all AIG nodes g_k, which occur in the linking polynomials $-s_i + g_k$ are eliminated. The variable g_k which links s_{2n-1} is not eliminated, as it acts as a carry. Since the coefficient of the variables g_k in the linking polynomials is 1, only the addition rule is required for D-reduction. We have $2n - 1$ such rules.

In the full- and half-adders the variables with only one parent get eliminated, that is g_1 and g_4 in Fig. 5b and g_1 in Fig. 5c. In total $2(n^2 - 2n) + n = 2n^2 - 3n$ variables are eliminated and each of these eliminations requires two rules. Hence "Local-Elimination" totally requires $2n - 1 + 2(2n^2 - 3n) = 4n^2 - 4n - 1$ proof rules.

For "Incremental-Reduction" we need to consider the multiplication rules as well as the summation rules. After variable elimination $8n^2 - 9n - (2n - 1) - (2n^2 - 3n) = 6n^2 - 8n + 1$ polynomials remain in $G(C)$. Each of them, except for the single polynomial $-s_0 + a_0 b_0$ in S_0 produces a multiplication rule. Thus $6n^2 - 8n$ multiplication rules are generated.

The $6n^2 - 8n$ factors plus the polynomial $-s_0 + a_0 b_0$, are summed up slice-wise to produce the carry recurrence relations. The sum of these carry recurrence relations produces the multiplier specification. Thus $6n^2 - 8n$ additions are necessary. Collecting all the generated proof rules leads to the final number of $4n^2 - 4n - 1 + 2(6n^2 - 8n) = 16n^2 - 20n - 1$ proof rules. □

Theorem 3. The degree of the PAC proof of n-bit btor-multipliers is 3.

Proof. The degree of the polynomials in the initial constraints set is at most 2, since the degree of the polynomials induced by AIG nodes is 2 and the linking polynomials have degree 1.

The degree of the PAC proof can only increase in multiplication rules. In the remainder of the proof we will heavily use the annotation of the variables as in Fig. 5b and Fig. 5c.

In "Local Elimination" we eliminate g_1 and g_4 from the full-adders and g_1 from the half-adders. As they have the same internal structure, we only discuss elimination of g_1 from half-adders. To eliminate g_1, the polynomial $p = -s + (1 - c)(1 - g_1)$ is D-reduced by $q = -g_1 + (1 - x)(1 - y)$. Hence "Divide-by-lm" of Alg. 5 yields $p_d = c - 1$. Consequently, the resulting polynomial of multiplying q and p_d is $p_m = -g_1 c + g_1 + xyc - xy - xc + x - yc + y + c - 1$ and has degree 3.

In the slicing algorithm btor-multipliers are partitioned in such a way, that all nodes of a full- and half-adder belong to the same slice. Thus the internal nodes of full- and half-adders are reduced in sequence, which has the consequence, that summing up the factored gate polynomials of internal adder nodes yields the adder specifications $2(1 - c) + s = x + y + z$ for full-adders and $2c + s = x + y$ for half-adders. We use this observation to determine the degree of the factors.

We first discuss half-adders. After local elimination of g_1 half-adders are modeled by the polynomials $-s + (c - 1)(xy - x - y)$ and $-c + xy$. The following multiplication rules are generated during incremental reduction. The constant α depends on the slice in which the half-adder belongs. Both conclusion polynomials have degree 3 and adding them yields the specification of a half-adder.

```
*: -s+(c-1)(xy-x-y), α,          -αs+αcxy-αcx-αcy-αxy+αx+αy;
*: -c+xy,            α(xy-x-y+2), -αcxy+αcx+αcy-2αc+αxy;
```

For full-adders the following factors are generated. All of them have at most degree 3.

```
*: -s+(g5-1)(g3x-g3-x), α,      -αs+αg5g3x-αg5x-αg5g3-αg3x+αg3+αx;
*: -c+(1-g5)(1-g2),     -2α,    2αc-2αg5g2+2αg5+2αg2-2α;
*: -g5+g3x,   α(g3x-g3-2g2-x+2), -αg5g3x+αg5g3+2αg5g2+αg5x-2αg5-2αg3g2x+αg3x;
*: -g3+(g2-1)(yz-y-z), α(-2g2x+1), 2αg3g2x-αg3+αg2yz-αg2y-αg2z-αyz+αy+αz;
*: -g2+yz,    α(yz-y-z+2),    -αg2yz+αg2y+αg2z-2αg2+αyz;
```

All polynomials, which model partial products are only multiplied by constants. Thus we never generated a polynomial which has a degree larger than 3. □

In contrast to proof length and degree we are only able to determine an upper bound for proof size.

Theorem 4. The proof size of n-bit btor-multipliers is in $\mathcal{O}(n^2 \log(n))$.

Proof. As in the previous proofs we distinguish between "Local Elimination" and "Incremental Reduction". Eliminating g_k from the $2n$ linking polynomials $-s_i+g_k$ needs only one addition. The conclusion polynomial has at most 5 monomials, since each gate constraint contains at most 5 monomials.

Elimination of g_1 and g_4 in the full-adders and g_1 in the half-adders produces one multiplication rule and one addition rule. In the proof of Thm. 3 we listed the conclusion polynomial p_m of the multiplication, which has size 10. Adding p_m to $-s+(1-g_1)(1-c)$ yields a polynomial with 7 monomials. Since we eliminate two variables from each full-adder and one variable from each half-adder, we eliminate $2n^2-3n$ variables. Each elimination produces 17 monomials. Thus "Local Elimination" produces at most $5(2n)+17(2n^2-3n) = 34n^2+41n$ monomials.

In "Incremental Reduction" we need to consider the multiplication rules as well as the addition rules which add up the polynomials slice-wise and then totally to gain the word-level specification.

The n^2 polynomials defining the partial products are multiplied by constants 2^i, thus each conclusion polynomial has 2 monomials. We have already written down each multiplication rule for the full- and half-adders in the proof of Thm. 3. Counting the monomials yields 32 monomials for each full-adder and 12 monomials for each half-adder. Thus in total $2n^2+32(n^2-2n)+12n = 34n^2-52n$ monomials are needed in the multiplication rules.

After the factors are generated, they are added up in a tree-like approach, as discussed at the end of Sect. 5. If m polynomials are added, the depth of the corresponding addition tree is $\lceil \log(m) \rceil + 1$.

First the polynomials within one slice are summed up. The biggest slice is S_{n-1}, which contains $n-2$ full-adders, 1 half-adder and n partial products. Thus in total $6n-8$ polynomials are added. For simplicity we drop the constant and assume $6n$ polynomials are added. The depth of the tree is $\lceil \log(6n) \rceil + 1 < \lceil \log(6) \rceil + \lceil \log(n) \rceil + 1 < \lceil \log(n) \rceil + 4$.

It can be seen in the proof of Thm. 3, that each polynomial contains at most 8 monomials. Thus the initial layer of the addition tree has at most $48n$ monomials. Let us assume adding two polynomials does not cancel any monomials. Thus in layer i of the addition tree, the polynomials have $2^i \cdot 8$ monomials. Since each layer has $\frac{1}{2^i}(6n)$ polynomials, the total number of monomials for each layer is $48n$. Adding up one slice produces at most $48n(\lceil \log(n) \rceil + 4) = 48n\lceil \log(n) \rceil + 192n$ monomials. Since we have $2n$ slices, we have at most $96n^2\lceil \log(n) \rceil + 384n^2$ monomials.

We add up these carry recurrence relations to gain the word-level specification. We have $2n$ carry recurrence relations and each of them contains one monomial for the output variable s_i, at most $n-1$ monomials for the incoming carries and $n-1$ monomials for the outgoing carries and at most n partial products and one constant monomial. Adding two consecutive carry recurrence relations cancels the matching outgoing and incoming carries. Thus after adding two initial polynomials, the resulting polynomials contains 2 monomials for the output bits, at most $2n-2$ monomials related to carries and at most $2n$ partial products and a constant. Let $m = \lceil \log(2n) \rceil \leq \lceil \log(n) \rceil + 1$. We have $m+1$ addition layers and each layer contains $\frac{2n}{2^i}$ polynomials. Thus the upper bound of monomials is

$$\sum_{i=0}^{m} \frac{2n}{2^i}(\overbrace{2^i}^{\text{output}} + \overbrace{n-1}^{\text{carry in}} + \overbrace{n-1}^{\text{carry out}} + \overbrace{2^i n}^{\text{p.products}} + \overbrace{1}^{\text{constant}}) = 2n \sum_{i=0}^{m} (n+1) + 2n(2n-1) \sum_{i=0}^{m} \frac{1}{2^i} <$$

$$2n(n+1)(\lceil \log(n) \rceil + 2) + 4n^2(2 - \frac{1}{4n}) = 2n^2\lceil \log(n) \rceil + 2n\lceil \log(n) \rceil + 12n^2 + 3n.$$

Altogether our upper bound yields the polynomial $98n^2\lceil \log(n) \rceil + 2n\lceil \log(n) \rceil + 464n^2 - 8n$ and thus we are in $\mathcal{O}(n^2 \log(n))$. □

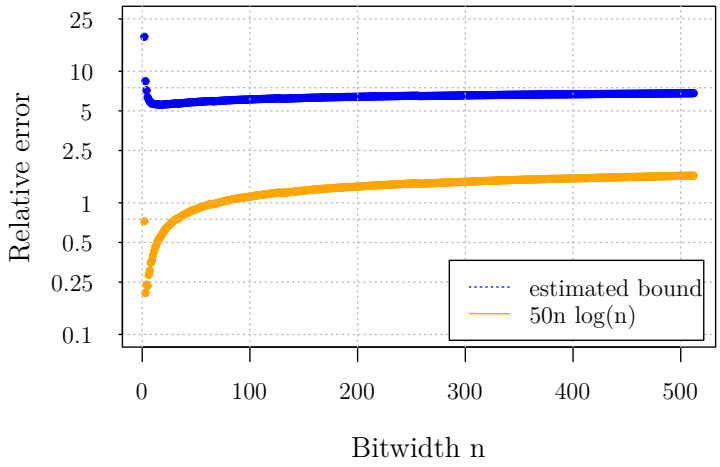


Figure 6: Proof size for $n = [2, 512]$.

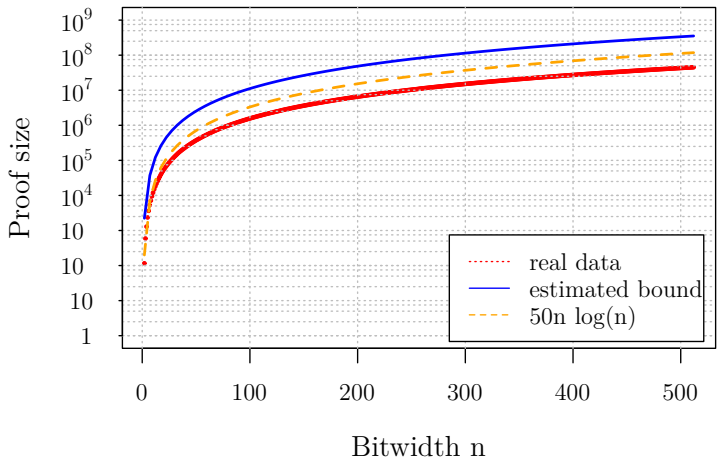


Figure 7: Relative errors of upper bounds for $n = [2, 512]$.

We could clearly improve this bound, as we now considered all $2n$ slices to contain the same number of polynomials as the largest slice. Furthermore monomials do cancel when the polynomials within a slice are summed up. The real proof size as well as the estimated upper bound can be seen in Fig. 6. We further added the function $50n^2 \log(n)$ in the plots, which also seems to be sufficient as an upper bound. In Fig. 7 we show the relative errors of the upper bounds.

7 Conclusion

In this paper we presented our tool AMULET, a state-of-the-art tool to automatically verify and certify the correctness of large gate-level integer multipliers. We gave an introduction into the problem of arithmetic circuit verification and discussed our state-of-the-art solving method which combines SAT and computer algebra. Certain parts, more precisely complex final stage adders, of the multiplier are detected and replaced by simple ripple-carry adders. The correctness of the replacement is checked by SAT solvers and the rewritten multiplier is verified using computer algebra. We presented details of the underlying algorithms to detect final stage adders and rewrite the multipliers, originally introduced in [14]. Furthermore we reconsidered our incremental verification algorithm and discussed the procedure of generating proof certificates. For one specific simple type of multipliers we showed that we are able to generate proof certificates with length in $\mathcal{O}(n^2)$ and size in $\mathcal{O}(n^2 \log(n))$. In the future we want to be able to extend our methods to synthesized multipliers where technology mapping is applied. Investigating floating points and other word-level operators is interesting future work too.

References

[1] Paul Beame and Vincent Liew. Toward verifying nonlinear integer arithmetic. *J. ACM*, 66(3):22:1–22:30, June 2019.

[2] Thomas Becker, Volker Weispfenning, and Heinz Kredel. *Gröbner Bases*. Springer, 1993.

[3] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, 1965.

[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] Matthew Clegg, Jeffery Edmonds, and Russell Impagliazzo. Using the groebner basis algorithm to find proofs of unsatisfiability. In *STOC*, pages 174–183. ACM, 1996.

[6] Luís Cruz-Filipe, João Marques-Silva, and Peter Schneider-Kamp. Formally verifying the solution to the boolean pythagorean triples problem. *J. Autom. Reasoning*, 63(3):695–722, 2019.

[7] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. SINGULAR 4-1-0. http://www.singular.uni-kl.de, 2016.

[8] Torbjörn Granlund et al. GNU MP: The GNU Multiple Precision Arithmetic Library, 2016. http://gmplib.org/.

[9] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *ISAIM*, 2008.

[10] Allen Van Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.

[11] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *FMCAD 2013*, pages 181–188. IEEE, 2013.

[12] Warren A. Hunt, Matt Kaufmann, J Strother Moore, and Anna Slobodova. Industrial hardware and software verification with acl2. *Philos. Trans. Royal Soc. A*, 375:20150399, 10 2017.

[13] Daniela Kaufmann, Armin Biere, and Manuel Kauers. Incremental Column-wise verification of arithmetic circuits using computer algebra. *Formal Methods in System Design*, Feb 2019.

[14] Daniela Kaufmann, Armin Biere, and Manuel Kauers. Verifying Large Multipliers by Combining SAT and Computer Algebra. In *FMCAD 2019*, To appear.

[15] Daniela Kaufmann, Manuel Kauers, Armin Biere, and David Cok. Arithmetic Verification Problems Submitted to the SAT Race 2019. In *Proc. of SAT Race 2019*, 2019. Submitted.

[16] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE TCAD*, 21(12):1377–1394, 2002.

[17] Alireza Mahzoon, Daniel Große, and Rolf Drechsler. PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers. In Iris Bahar, editor, *ICCAD*, page 129. ACM, 2018.

[18] Alireza Mahzoon, Daniel Große, and Rolf Drechsler. RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. In *Design Automation Conf.*, 2019. In press.

[19] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , BtorMC and Boolector 3.0. In *Computer Aided Verification, CAV*, volume 10981 of *LNCS*, pages 587–595. Springer, 2018.

[20] Behrooz Parhami. *Computer Arithmetic - Algorithms and Hardware designs*. Oxford University Press, 2000.

[21] Daniela Ritirc, Armin Biere, and Manuel Kauers. A Practical Polynomial Calculus for Arithmetic Circuit Verification. In *SC-Square Workshop 2018*, pages 61–76. CEUR-WS, 2018.

[22] Wolfram Research, Inc. Mathematica, 2016. Version 10.4.

[23] Cunxi Yu, Maciej J. Ciesielski, and Alan Mishchenko. Fast Algebraic Rewriting Based on And-Inverter Graphs. *IEEE TCAD*, 37(9):1907–1911, 2018.