Column-Wise Verification of Multipliers Using Computer Algebra

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

***Abstract*—Verifying arithmetic circuits, and most prominently multipliers, is an important problem but in practice still requires substantial manual effort. Recent work tries to solve this issue using techniques from computer algebra. The most effective approach uses polynomial reasoning over pseudo boolean polynomials. In this paper we give a rigorous formalization of this approach and present a new column-wise verification technique for the correctness of gate-level multipliers which does not require the reduction of a full word-level specification. We formally prove soundness and completeness of our technique, making use of our precise formalization. Our experiments show that simple multipliers can be verified efficiently by using off-the-shelf computer algebra tools, while more complex and optimized multipliers require more sophisticated techniques. Further, our paper independently confirms the effectiveness of previous related work. We make all benchmarks and tools publicly available.**

I. Introduction

Formal verification of arithmetic circuits is motivated by the necessity to avoid issues like the famous Pentium FDIV bug, which is reported to have cost Intel almost half a billion dollar. There have been many attempts since then to verify such circuits, but even today verifying designs with arithmetic parts is not considered to be fully automated. For instance, a common approach is to black-box multipliers and then verify them separately. This might also require insight into the multiplier design, which has to be communicated to the verification tool. Commercial tools can not fully automatically handle full-sized multipliers [24] or huge multipliers occurring in cryptographic circuits. In this paper we will focus, as a first step, on the simplest but also most important arithmetic circuit verification problem of verifying multipliers.

This lack of automation was a common conclusion in three plenary talks at the joint FMCAD'15 and SAT'15 conferences in Austin in 2015, by Anna Slobodova on formal verification of processors, Aaron Tomb on verifying cryptographic circuits, and, from the academic side, Priyank Kalla on methods for data path verification. In order to stimulate research into this direction, particularly the development of fast SAT solving techniques for arithmetic circuit verification, we collected a large set of such benchmarks, generated and submitted CNF encodings of these problems to the SAT 2016 competition and made them publicly available [4]. The competition results confirmed that miters of even small multipliers pose a real challenge to SAT solvers.

The weak performance of SAT solvers on these benchmarks lead to the conjecture that verifying miters of multipliers and other ring properties after encoding them into CNF needs exponential sized resolution proofs [5], which would imply exponential run-time of CDCL SAT solvers. Surprisingly, however, this conjecture was recently answered negatively [2]. Such ring properties do admit polynomial resolution proofs. However, proof search is non-deterministic. Thus this theoretical result still needs to be transferred into practical SAT solving. The complexity bounds on proof size given in [2] involve polynomials of high degree too.

The first technique which was shown to be able to have prevented the Pentium bug was based on decision diagrams, precisely on binary moment diagrams (BMDs) [10] and variants [11]. While common (gate-level) BDDs are exponential in size for multipliers [6], BMDs remain linear in the number of the input bits of a multiplier (using edge weights). However, the BMD approach is not robust, in the sense that it still requires structural knowledge of the multipliers to determine the order in which BMDs are built, which has tremendous influence on performance. Actually only a row-wise backward substitution approach seems to be feasibly [9], which in addition assumes a simple carry-save-adder (CSA) design.

Recent algebraically inspired techniques [12], [28] based on so-called function-extraction also fail for even slightly optimized multiplier designs. On the positive side, this technique is able to handle very large clean multipliers.

In even more recent work [24] substantial progress was made. The authors use a dedicated polynomial reduction engine and also gave various optimizations (discussed further down), which made their algebraic technique scale to large non-trivial multiplier designs of various architectures [16] (called AOKI benchmarks in the following) even with and without Booth reencoding. It is still unclear however, whether their technique is robust under synthesis or technology mapping. Their arguments for soundness and completeness are rather imprecise. Their tool is not available, nor details about the experiments. Benchmarks have not been published either.

There is a substantial amount of previous work for arithmetic circuit verification. We focus on comparing our approach to the currently most successful techniques for verifying multipliers, which all are using some form of algebraic reasoning [28], [24]. For an up-to-date discussion of related work and a more comprehensive list see the recent article [28].

Supported by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), Y464-N18, SFB F5004.

II. Algebra

Following [21], [23], [12], [28], we model the behavior of circuits using multivariate polynomials. There will be a variable for every input and every output of each gate, and the specification of each gate is translated into a polynomial relation among these variables. All these polynomials together form a description of the circuit, and we will prove the correctness of a given circuit by showing that the desired relation between input and output is implied by the polynomials that describe the circuit on the gate level.

The appropriate formalism for such a reasoning is the theory of Gröbner bases [8], [13]. Basic facts are:

- $\mathbb{Q}[X] = \mathbb{Q}[x_1, \dots, x_n]$ denotes the ring of polynomials in variables $x_1, \dots, x_n$ with coefficients in the field $\mathbb{Q}$.
- A *term* (or *power product*) is a polynomial of the form $x_1^{e_1} \cdots x_n^{e_n}$ for certain $e_1, \dots, e_n \in \mathbb{N}$. A *monomial* is a constant multiple of a term.
- Fix an order $\leq$ on the set of terms such that for all terms τ, σ_1, σ_2 we have $1 \leq \tau$ and $\sigma_1 \leq \sigma_2 \Rightarrow \tau\sigma_1 \leq \tau\sigma_2$.
- Every polynomial $p \neq 0$ contains only finitely many terms, the largest of which (w.r.t. the chosen order $\leq$) is called the *leading term* and denoted by $\operatorname{lt}(p)$.
- If $p = c\tau + \cdots$ and $\operatorname{lt}(p) = \tau$, then $\operatorname{lc}(p) = c$ is called the *leading coefficient* and $\operatorname{lm}(p) = c\tau$ is called the *leading monomial* of p.
- A nonempty subset $I \subseteq \mathbb{Q}[X]$ is called an *ideal* if $\forall\, p, q \in I : p + q \in I$ and $\forall\, p \in \mathbb{Q}[X]\ \forall\, q \in I : pq \in I$.
- If $I \subseteq \mathbb{Q}[X]$ is an ideal, then a set $\{p_1, \dots, p_m\} \subseteq \mathbb{Q}[X]$ is called a *basis* of I if $I = \{q_1p_1 + \cdots + q_mp_m \mid q_1, \dots, q_m \in \mathbb{Q}[X]\}$, i.e., if I consists of all the linear combinations of the p_i with polynomial coefficients.
- A basis $\{g_1, \dots, g_n\}$ of an ideal $I \subseteq \mathbb{Q}[X]$ is called a *Gröbner basis* (w.r.t. the fixed order $\leq$) if the leading term of every nonzero element of I is a multiple of (at least) one of the leading terms $\operatorname{lt}(g_1), \dots, \operatorname{lt}(g_n)$.
- Every ideal of $\mathbb{Q}[X]$ has a Gröbner basis, and there is an algorithm which, given an arbitrary basis of an ideal, computes a Gröbner basis of it.

The theory of Gröbner bases offers a decision procedure for the ideal membership problem: given a polynomial $q \in \mathbb{Q}[X]$ and a basis $\{p_1, \dots, p_m\} \subseteq \mathbb{Q}[X]$, it is a priori not obvious how to check whether q belongs to the ideal generated by $p_1, \dots, p_m$. However, if $\{p_1, \dots, p_m\}$ is a Gröbner basis, then the question can be answered using a multivariate version of polynomial division with remainder. It can be shown that when G is a Gröbner basis, then q belongs to the ideal generated by G iff the remainder of division of q by G is zero.

Example 1.

1) Consider $q = x^2 + 4x + 3$, $p = x + 1 \in \mathbb{Q}[x]$. Since $x^2+4x+3 = (x+3)(x+1)+0$, it follows that q belongs to the ideal I generated by $x+1$ in $\mathbb{Q}[x]$. On the other hand, taking $\tilde{q} = x^2+4x+5$, division with remainder gives $x^2+4x+5 = (x+3)(x+1)+2$, and thus $\tilde{q} \notin I$.

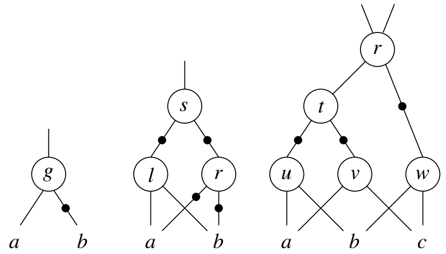

Fig. 1. AIGs [20] used in Example 1 and Sect. IV.

2) For the AIG [20] on the left of Fig. 1, we have the relation $g = a(1-b)$ for all $a, b, g \in \{0,1\}$. Furthermore, we always have $g(g-1) = a(a-1) = b(b-1) = 0$ for all $a, b, g \in \{0,1\}$. To show that we always have $gb = 0$, it is therefore enough to check if the polynomial $gb \in \mathbb{Q}[g,a,b]$ belongs to the ideal $I \subseteq \mathbb{Q}[g,a,b]$ generated by

$$\{-g + a(1-b), g(g-1), a(a-1), b(b-1)\}.$$

Multivariate polynomial division yields

$$gb = (-b)\,(-g + a(1-b)) + (-a)\,b(b-1) + \overset{\overset{\text{remainder}}{\downarrow}}{0},$$

therefore $gb \in I$ and thus $gb = 0$ in the left AIG of Fig. 1.

As illustrated in the second example, we can view an ideal $I \subseteq \mathbb{Q}[X]$ as an equational theory, with a basis $\{p_1, \dots, p_m\}$ as its set of axioms. Indeed, the ideal I generated by $p_1, \dots, p_m$ contains exactly those polynomials q for which the equation "$q = 0$" can be deduced from the assumptions "$p_1 = \cdots = p_m = 0$" through repeated application of the rules $u = 0 \land v = 0 \Rightarrow u + v = 0$ and $u = 0 \Rightarrow uw = 0$ (compare the two defining properties for ideals quoted above).

We will need a few more facts about Gröbner bases and multivariate polynomial division.

Lemma 1.

1) Let $q \in \mathbb{Q}[X]$ and $P = \{p_1, \dots, p_m\} \subseteq \mathbb{Q}[X]$. The remainder r of the division of q by P is a polynomial such that $q - r$ is in the ideal generated by P and r is *reduced* w.r.t. P, which means it does not contain any term that is a multiple of one of the leading terms $\operatorname{lt}(p_1), \dots, \operatorname{lt}(p_m)$.

2) Let $G \subseteq \mathbb{Q}[X] \setminus \{0\}$, and define

$$\operatorname{spol}(p,q) := \operatorname{lcm}(\operatorname{lt}(p), \operatorname{lt}(q)) \left(\frac{p}{\operatorname{lm}(p)} - \frac{q}{\operatorname{lm}(q)} \right)$$

for all $p, q \in \mathbb{Q}[X] \setminus \{0\}$, with lcm the least common multiple. Then G is a Gröbner basis if and only if the remainder of the division of $\operatorname{spol}(p,q)$ by G is zero for all pairs $(p,q) \in G \times G$.

3) If $p, q \in \mathbb{Q}[X] \setminus \{0\}$ are such that their leading terms $\operatorname{lt}(p), \operatorname{lt}(q)$ have no variables in common, then the division of $\operatorname{spol}(p,q)$ with $\{p,q\}$ has remainder zero.

Proof. 1) is Prop. 1 in Chap. 2 §6 of [13]; 2) is Thm. 6 in Chap. 2 §6 of [13]; 3) is Prop. 1 in Chap. 2 §10 of [13]. □

III. Ideals associated to circuits

We consider circuits with $2n$ inputs $a_0, \dots, a_{n-1}$ and $b_0, \dots, b_{n-1}$, $2n$ outputs $s_0, \dots, s_{2n-1}$, and a number of logical gates. The output of some gate may be input to some other gate, but cycles are not allowed. In addition to the variables for input and output, we also associate one variable to each internal edge of the circuit, say $g_1, \dots, g_k$. By R we denote the ring $\mathbb{Q}[a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1}, g_1, \dots, g_k, s_0, \dots, s_{2n-1}]$.

The semantics of the circuit gates imply polynomial relations among these variables, such as the following ones:

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

We also have the relations $u(u-1) = 0$ for each variable u, because the circuit operates with boolean values.

Since logical gates are functional, the values of all the variables $g_1, \dots, g_k, s_0, \dots, s_{2n-1}$ in a circuit are uniquely determined as soon as $a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1} \in \{0, 1\}$ are fixed. This motivates the following definition.

Definition 1. Let C be a circuit.

1) A polynomial $p \in R$ is called a *polynomial circuit constraint (PCC)* for C if for every choice of

$$(a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1}) \in \{0, 1\}^{2n}$$

and resulting values $g_1, \dots, g_k, s_0, \dots, s_{2n-1}$ implied by the gates of C the substitution of these values into p gives zero.

2) The set of all PCCs for C is denoted by $I(C)$.

It is easy to see that $I(C)$ is in fact an ideal of R. By definition, this ideal contains all the relations that hold among the values at the different points in the circuit. In particular, it "knows" everything about how input and output are related. Therefore, the circuit fulfills a certain specification if and only if the polynomial relation corresponding to the specification is contained in $I(C)$. This motivates the next definition.

Definition 2. A circuit C is called a *multiplier* if

$$\sum_{i=0}^{2n-1} 2^i s_i - \left(\sum_{i=0}^{n-1} 2^i a_i \right) \left(\sum_{i=0}^{n-1} 2^i b_i \right) \in I(C).$$

Checking whether a given circuit C is a multiplier thus reduces to an ideal membership test. Definition 1 does not provide us with a basis of $I(C)$, so Gröbner basis technology is not directly applicable. However, we can deduce at least some elements of $I(C)$ from the semantics of circuit gates.

Definition 3. Let C be a circuit. Let $G \subseteq R$ be the set which contains for each gate of C the corresponding polynomial of (1) (with u, v, w replaced by the variables of the edges attached to the gate), as well as the polynomials $a_i(a_i - 1)$ and $b_i(b_i - 1)$ for $0 \leq i < n$, called *input field polynomials*. Then the ideal generated by G in R is denoted by $J(C)$.

As a basis of $J(C)$ is explicitly known, we can decide membership using Gröbner bases. Consider a verifier for circuits which checks for a given C and a given specification polynomial p whether p belongs to $J(C)$. Because of $J(C) \subseteq I(C)$, such a verifier is certainly sound. In order to prove that it is also complete, we need to show $J(C) \supseteq I(C)$. For doing so, we recall a crucial observation which for instance already appears in [26], [21].

Theorem 1. Let C be a circuit, and let G be as in Def. 3. Let $\leq$ be a lexicographic term order for a variable order such that the variable attached to the output edge of a gate is always greater than the variables attached to the input edges of that gate. Then G is a Gröbner basis with respect to $\leq$.

Proof. By the constraint on the term order and the form of the equations (1), the leading term of each gate polynomial is simply the output variable of the corresponding gate. Further, the leading terms of the polynomials $a_i(a_i - 1)$ and $b_i(b_i - 1)$ are a_i^2 and b_i^2. Therefore, by part 3 of Lemma 1, division of $\mathrm{spol}(p, q)$ by $\{p, q\}$ gives the remainder zero for any choice $p, q \in G$. Then, since $\{p, q\} \subseteq G$ for all $p, q \in G$, also division of $\mathrm{spol}(p, q)$ by G gives the remainder zero for all $p, q \in G$, and then, by part 2 of Lemma 1, the claim follows. □

Theorem 2. For all acyclic circuits C, we have $J(C) = I(C)$.

Proof. "$\subseteq$" (soundness) Clear by definition of $J(C)$.

"$\supseteq$" (completeness) Let $p \in I(C)$. We have to show that $p \in J(C)$. Since C is acyclic, there is a way to order the variables such as to meet the requirement of Thm. 1. Let r be the remainder of the division of p by G, where G is the Gröbner basis of Thm. 1. Then $p - r \in J(C)$ by part 1 of Lemma 1, so $r \in J(C) \iff p \in J(C)$. Furthermore, $p \in I(C)$ and $p - r \in J(C) \subseteq I(C)$ implies $r \in I(C)$. It is therefore sufficient to show that $r \in J(C)$.

By the choice of the term order and the observations made in the previous proof about the leading terms in G, part 1 of Lemma 1 also implies that r only contains input variables $a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1}$, and none of them appears with degree greater than one. At the same time, since $r \in I(C)$, all the evaluations of r for all choices $a_i, b_j \in \{0, 1\}$ are zero.

We show that $r = 0$, and thus $r \in J(C)$. Suppose $r \neq 0$. Let m be a monomial of r with a minimal number of variables, which includes the case where m is constant. Since exponents are at most one, the set of variables of monomials in r differ by at least one variable. Now choose a_i (b_j) to evaluate to 1 iff $a_i \in m$ ($b_j \in m$). By this choice all monomials of r except m vanish (evaluate to zero). Thus r evaluates to the (non-zero) coefficient of m, in contradiction to $r \in I(C)$. □

Let us conclude the theoretical part of this paper with the following simple but important observations.

First, $I(C)$ is by definition a so-called vanishing ideal. Therefore, the theorem implies that $J(C)$ is a radical ideal. This explains why ideal membership is sufficient for our purpose, and there is no need to use the stronger radical membership test (cf. Chap. 4 §2 of [13]).

Second, note that $I(C) = J(C)$ contains the set F of all field polynomials $x(x-1)$ for all variables x, not only for inputs, thus we may add them to G.

Third, in the standard Gröbner basis for gate-level circuits defined above in Def. 3 using Eqn. (1) all polynomials have a leading coefficient of ± 1 and thus during reduction never introduce any coefficient outside of $\mathbb{Z}$ (with non-trivial denominator). So all coefficient computations actually remain in $\mathbb{Z}$. This formally proves that dedicated implementations, e.g., those from [28], [24], used for determining ideal membership to verify properties of gate-level circuits, can rely on computation in $\mathbb{Z}$ only without loosing soundness nor completeness (assuming the same term order as in Thm. 1 is used).

Fourth, from a technical point of view, we do not need to use $\mathbb{Z}$ as coefficient ring if we employ computer algebra systems, but can simply use any field containing $\mathbb{Z}$, e.g., $\mathbb{Q}$. This actually speeds up the computation, since computer algebra systems are optimized for this case. In our experiments, using rational coefficients made a huge difference for Singular [14] (but did not have any effect in Mathematica [27]).

Fifth, given circuit C, checking that there exists an assignment to the inputs which yields a certain value, at an output is of course the same as (circuit) SAT, and thus NP complete:

Corollary 1. Checking ideal membership over $\mathbb{Q}[X]$ even in terms of a given Gröbner basis is co-NP-hard.

Similar results but for $\mathbb{Z}_2$ and $\mathbb{Z}$ instead of $\mathbb{Q}$ and without assuming a Gröbner basis can be found in [1], [18].

Finally, the last part in the proof of Thm. 2 allows us to determine a concrete input evaluation in case a polynomial g fails the membership test, e.g., an evaluation for which g does not vanish. In our application of multiplier verification these evaluations provide counter-examples, in case a circuit is determined not to be a multiplier (Alg. 1 returns *false*).

We claim that this section is a first simple and precise mathematical characterization of recent successful algebraic approaches [24], [28] to the verification of gate-level integer multipliers (without overflow), where we formally prove not only soundness but also completeness. Soundness corresponds to $I \subseteq J$ and completeness to $I \supseteq J$ in Thm. 2.

In previous work soundness and completeness was formally proven too but only for other polynomial rings, i.e., over $\mathbb{F}_{2^q}$ to model circuits implementing Galois-field multipliers [21], [23], or for polynomial rings over $\mathbb{Z}_{2^q}$ to model arithmetic circuit verification with overflow semantics [26].

In [28] soundness and completeness is discussed too, but instead of giving proofs only refers to [21], [23] which as discussed above uses coefficients in $\mathbb{F}_{2^q}$ and not $\mathbb{Z}$, the coefficient ring the approach [28] is actually working with.

IV. Optimizations

Following the argument of Cor. 1 in the previous section, simply reducing the specification in the constructed Gröbner basis may lead and in general has to lead (unless P = NP) to an exponential number of monomials in intermediate results. Thus in practice to use polynomial reduction to verify specific circuits tailored heuristics become very important.

To reduce the number of monomials in [24] a logic reduction rewriting scheme consisting of *XOR-Rewriting* and *Common-Rewriting* is proposed. It is further combined with eliminating monomials which fulfill certain *Vanishing-Constraints*. In the following we show how these techniques can be applied to computer algebra systems.

The technique of *XOR-Rewriting* [24] ensures that in the Gröbner basis all variables which do not correspond to an output nor input of an XOR-gate, nor primary input, nor output of the circuit, are eliminated from the Gröbner basis up-front.

We adopt this rewriting for AIGs by matching XOR patterns in the AIG which represents an XOR or XNOR, e.g., we find nodes of the form $s = \overline{(a \wedge b)} \wedge \overline{(\bar{a} \wedge \bar{b})}$. We then define the polynomial of the parent in terms of the grandchildren instead of the immediate children. For instance, in order to apply XOR-Rewriting in the middle AIG in Fig. 1 we only use the polynomial $-s + a + b - 2ab$ as definition for the XOR output instead of all the polynomials $-l + ab$, as well as $-r + (1-a)(1-b)$, and $-s + (1-l)(1-r)$. This removes defining polynomials for all children of XOR gates.

The technique of *Common-Rewriting* [24] eliminates all nodes which have exactly one parent. In the right AIG of Fig. 1 Common-Rewriting eliminates gates t, u, v, and w, assuming r occurs twice, but t, u, v and w only once. Thus r is directly expressed in terms of a, b, c. This technique is actually similar to what bounded variable elimination in SAT would do [15] after encoding a circuit to CNF by say Tseitin encoding. It would also eliminate all variables in the CNF representing gates in the circuit with only one parent [17].

In [24] an important optimization was a specific “vanishing rule”, called *XOR-AND Vanishing Rule*. This rule can be derived from the middle AIG in Fig. 1, a half adder, where l represents the carry (AND) and s represents the sum (XOR) of the two inputs. In a half adder both the carry bit l and the sum bit s can never be 1 at the same time. Thus $sl = 0$, and [24] suggests to remove monomials containing s and l immediately. We simulate the effect of this rule by searching for (negated) children or grand-children of certain AND-gates and adding appropriate polynomial constraints to our reduction basis.

V. Order

According to Thm. 1 the choice of the reverse topological term order does not influence the correctness of the procedure. However in [28] it is shown empirically that the number of monomials during the reduction process varies substantially for different reverse topological orders.

Given the planar two dimensional “shape” of multipliers, two approaches of ordering are quite natural, namely a row-wise approach and a column-wise approach. Basically the idea is to partition the gates into *slices*, which are totally ordered, i.e., row-wise or column-wise, and then order the gates within a slice (row or column) topologically. The combined total order has to be topological, which then gives a valid term order and thus a Gröbner basis according to Thm. 1.

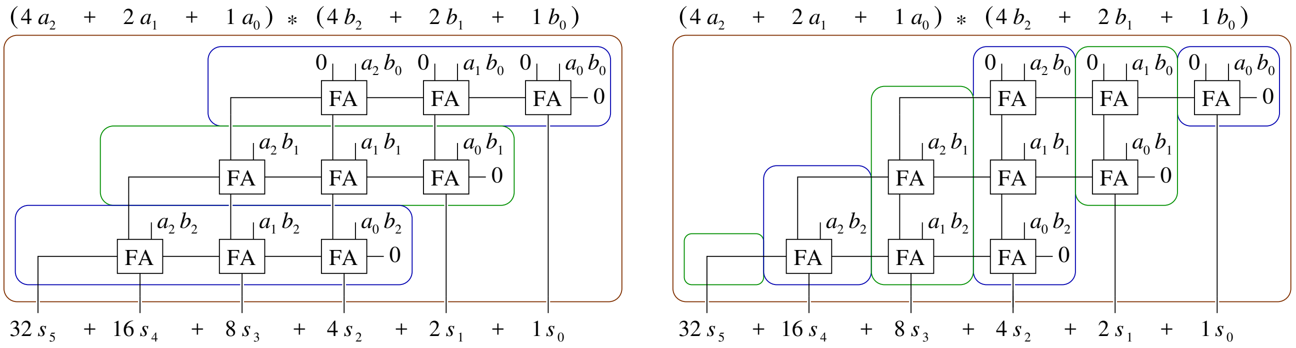
(4 a2 + 2 a1 + 1 a0) * (4 b2 + 2 b1 + 1 b0)
0
a2 b0
a1 b0
a0 b0
a2 b1
a1 b1
a0 b1
a2 b2
a1 b2
a0 b2
FA
0
32 s5 + 16 s4 + 8 s3 + 4 s2 + 2 s1 + 1 s0
(4 a2 + 2 a1 + 1 a0) * (4 b2 + 2 b1 + 1 b0)
32 s5 + 16 s4 + 8 s3 + 4 s2 + 2 s1 + 1 s0

Fig. 2. Classical row-wise slicing (left) versus our column-wise slicing approach (right) for clean 3-bit input (6-bit output) CSA multiplier.

The idea of the row-wise approach is to order the gates according to their backward level. The intuition of row-wise slicing is outlined in the left side of Fig. 2. It shows how full adders are partitioned in a "clean" (CSA) multiplier. Informally, we call a multiplier without gate synthesis, nor mapping and where the XOR-gates and the half/full adders can easily be identified, as *clean*. If a multiplier is not clean, it is called *dirty*. Thus the AOKI benchmarks [16], [24] already discussed in the introduction are considered to be dirty.

Previous papers [28], [10] use a row-wise approach. In [28] gates are ordered by the logic level seen from the circuit inputs. In [10] the order is only given for clean CSA multipliers, such that a word-level spec for a CSA step can be given. It is unclear how to apply this order to dirty multipliers, like the AOKI benchmarks. Unfortunately, the description of the order in [24] stays on a very high level. The tool is not available.

In the column-wise approach, cf. right side of Fig. 2, the multiplier is partitioned vertically, where each slice contains exactly one output bit. Our proposal is to use a column-wise order which gives a more robust incremental checking method.

VI. Column-Wise Checking

The goal of using a column-wise term order is to divide the problem into smaller more manageable sub-problems, which can be verified incrementally.

Definition 4. Let C be a circuit (as in Sect. III).

1) A sequence of $2n+1$ polynomials $C_0, \ldots, C_{2n}$ over the variables of C is called a *carry sequence* of *carry polynomials*.

2) For column i with $0 \leq i < 2n$ let $P_i = \sum_{k+l=i} a_k b_l$ be the *partial product sum* (of column i).

3) For $0 \leq i < 2n$, carry polynomial C_i and output s_i let

$$-C_i + 2C_{i+1} + s_i - P_i$$

denote the *carry recurrence relation* R_i for column i.

4) Then R_i *holds* on C if it vanishes in $I(C)$, i.e., $R_i \in I(C)$.

With these definitions we obtain an abstract theorem which can be used to verify multipliers independent how the carry sequence is actually constructed.

Theorem 3. Let C be a circuit where all carry recurrence relations hold as defined in Def. 4. Then C is a multiplier in the sense of Def. 2, if and only if $C_0 - 2^{2n}C_{2n} \in I(C)$.

Proof. By the condition of Def. 4, we have (modulo $I(C)$)

$$\begin{aligned}\sum_{i=0}^{2n-1} 2^i s_i &= \sum_{i=0}^{2n-1} 2^i (P_i + C_i - 2C_{i+1}) \\ &= \sum_{i=0}^{2n-1} 2^i P_i + \underbrace{\sum_{i=0}^{2n-1} (2^i C_i - 2^{i+1} C_{i+1})}_{C_0 - 2^{2n} C_{2n}} .\end{aligned}$$

It remains to show $\sum_{i=0}^{2n-1} 2^i P_i = (\sum_{i=0}^{n-1} 2^i a_i)(\sum_{i=0}^{n-1} 2^i b_i)$, which is a rather straight forward calculation. □

To obtain our column-wise checking algorithm we define slices incrementally. For each output bit s_i we determine its input cone, namely the gates which s_i depends on (cf. Fig. 3):

$$I_i := \{\text{gate } g \mid g \text{ is in input cone of output } s_i\}$$

We define slices S_i as the difference of consecutive cones I_i:

$$S_0 := I_0 \qquad S_{i+1} := I_{i+1} \setminus \bigcup_{j=0}^{i} S_j$$

Definition 5 (Sliced Gröbner Bases)**.** Let G_i be the set of polynomial representations of the gates in slice S_i, cf. Eqn. 1.

Algorithm 1: Multiplier Checking Algorithm

```
Input  : Circuit $C$ with sliced Gröbner bases $G_i$
Output: Determine whether $C$ is a multiplier
$C_{2n} \leftarrow 0$;
for $i \leftarrow 2n-1$ to $0$ do
    $C_i \leftarrow$ Remainder $(2C_{i+1} + s_i - P_i, \;\; G_i \cup F\;)$
end
return $C_0 = 0$
```

In Alg. 1 we start at the last output bit s_i with $i = 2n-1$. Then C_i is computed recursively by taking the remainder of $2C_{i+1} + s_i - P_i$ modulo the sliced Gröbner basis G_i and (all) field polynomials F in order to make sure that the carry

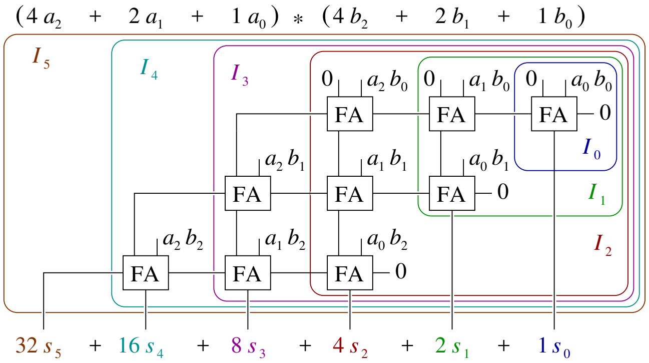
$(4\,a_2 \;+\; 2\,a_1 \;+\; 1\,a_0) * (4\,b_2 \;+\; 2\,b_1 \;+\; 1\,b_0)$
I_5
I_4
I_3
I_2
I_1
I_0
FA
0
$a_2\,b_0$
$a_1\,b_0$
$a_0\,b_0$
$a_2\,b_1$
$a_1\,b_1$
$a_0\,b_1$
$a_2\,b_2$
$a_1\,b_2$
$a_0\,b_2$
$32\,s_5 \;+\; 16\,s_4 \;+\; 8\,s_3 \;+\; 4\,s_2 \;+\; 2\,s_1 \;+\; 1\,s_0$

Fig. 3. Input cones of outputs to determine column slices.

recurrence relation R_i holds. Thus C_i is uniquely defined given the sum of the partial products P_i of column i, the output bit s_i and the previous carry polynomial C_{i+1}. It remains to fix the boundary carry polynomial C_{2n}. In our algorithm we actually always simply use $C_{2n} = 0$.

Theorem 4. Algorithm 1 returns *true* iff C is a multiplier.

Proof. By definition $R_i := -C_i + 2C_{i+1} + s_i - P_i$ vanishes on the ideal generated by $G_i \cup F$ which is a subset of the ideal generated by $G \cup F$ since $G_i \subseteq G$. Thus $R_i \in J(C) = I(C)$.

We can show inductively that C_i is reduced w.r.t. H_i with $H_i := \bigcup_{j \geq i}(G_j \cup F)$. This induction requires that s_i and P_i are reduced w.r.t. to H_{i+1} which holds due to the construction of the sliced Gröbner bases. With $H_0 = G \cup F$ we then get C_0 is reduced w.r.t. $G \cup F$ thus $C_0 = C_0 - 2^{2n} C_{2n} \in I(C) = J(C)$ iff $C_0 = 0$, which concludes the proof using Thm. 3. □

For incorrect multipliers Alg. 1 returns *false*, i.e., $C_0 \neq 0$. As described after Cor. 1 this easily yields a concrete counter-example. In this case it might further be possible to abort the algorithm earlier if partial products $a_k b_l$ of higher slices $k + l > i$ not occurring in S_j with $i < j$ remain in C_i.

VII. Engineering

Our tool AIGMulToPoly takes an AIG describing a circuit as input and produces output which can be passed to the computer algebra systems Mathematica [27] and Singular [14].

Algorithm 2: Outline of AIGMulToPoly

```
Input  : Circuit in AIG format
Output: File $f$ for computer algebra system
for $i \leftarrow 0$ to $2n-1$ do
    Define-Cones-of-Influence ();
    Merge ($S_i$);
    Promote ($S_i$);
    Levelize ($S_i$);
    Search-for-Common-Rewriting ($S_i$);
    Identify-Vanishing-Constraints ($S_i$);
end
$f \leftarrow$ Print to file;
```

For dirty multipliers slicing based on input cones, (Sect. VI), is not precise enough. It regularly happens, that gates are allocated to later slices, if they are not used to compute the output value of the slice. This frequently happens for carry outputs of full/half-adders (or combined carry outputs) and results in larger carry polynomials C_i than necessary.

To avoid this performance issue we eagerly move gates between slices, in a kind of peephole optimization, which makes sure that the overall number of carries decreases:

Definition 6. We define those gates in S_j used as children of gates in slice S_i with $i > j$ as *carries* of S_j.

The following technique reduces the support of carry polynomials increasing the chances for cancellation of monomials.

Merge: Whenever we find an AND-gate g (not matched to be an XOR- gate) in slice S_i with children l, r in smaller slices S_j and S_k, we move g back to S_l with $l = \max(j, k)$. The procedure is depicted on the left side of Fig. 4. Thus after merging g, the gates l, r are less likely to be carry variables any more. We apply merging repeatedly until completion and S_l and S_i are updated after each application.

In some multipliers it happens that a gate g in the carry depends on two other gates in the carry. We decrease the number of carries by promoting g to the next bigger slice:

Promote: We search for gates g in slice S_{i-1} with again exactly one parent, which in addition is required to be part of some larger slice S_j where $j \geq i$. Furthermore the children of g also have to be in slice S_{i-1} and have at least one parent in some later slice S_j with $j \geq i$. We decrease the number of carries by promoting g to slice S_i, cf. right side of Fig. 4.

A gate g which is merged can not be promoted back to its original slice, because the requirements for the children of g differ. This prevents cyclic rule applications. After merging and promoting, the association of gates to slices is fixed. We order the gates in a slice by levelization from inputs.

In order to simulate Common-Rewriting, we factor out from S_i the set U_i of "unique gates", i.e., all gates g of S_i not used in another slice with exactly one parent in slice S_i. Polynomials of gates which remain in S_i and depend on gates in U_i are reduced first by polynomials of gates in U_i and field polynomials F before computing the remainder in Alg. 1.

As last step we search for Vanishing Constraints in S_i, namely gate products which always evaluate to zero, e.g., Example 1. We store such constraints in a corresponding set V_i and during remainder computation reduce against elements of V_i too. Because of Thm. 2, we can add these polynomials to the ideal without violating the Gröbner basis property.

Finally, in AIGMulToPoly the optimization of "XOR-Rewriting" is handled implicitly during printing by producing polynomials for XOR-gates instead of AND-gates.

All optimizations either maintain the crictical criteria of keeping the reduction order topologically sorted, add vanishing constraints of the circuit ideal, or are standard techniques used in computer algebra, e.g., autoreduction, and thus do not affect correctness of our claims.

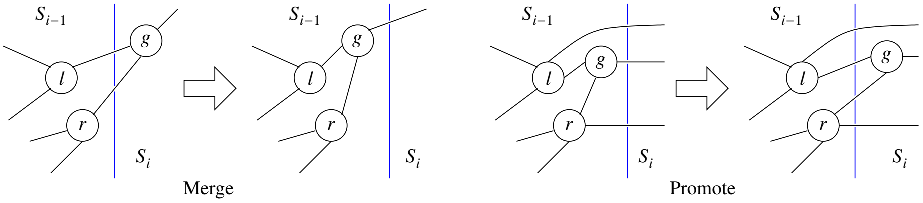

Fig. 4. Locally optimizing number of carries (gates used in later slices) by moving gates backward (Merge) and forward (Promote). Inputs are on the left, outputs on the right of AND-gates, which is the more common order used in visualizing circuits, but reversed compared to the column order in Fig. 2.

VIII. Experiments

As in previous work we focus on (integer) multipliers with two n-bit vectors as inputs and $2n$ output bits. In [28], [12] the authors used clean CSA multipliers, handcrafted from [19], for verifying their results. In [24] several architectures from the AOKI benchmarks are used in their experiments. In our experiments we use the multiplier types "btor", "sp-ar-rc" and "abc". The "btor" benchmarks are generated from Boolector [22] and can be considered as clean multipliers. The "sp-ar-rc" multipliers are part of the AOKI benchmarks [16] and can be considered as dirty multipliers. The "abc" benchmarks are generated with ABC [3]. The different versions of synthesis and technology mapping should be the same as in [28], [12].

We used a standard Ubuntu 16.04 Desktop machine with Intel i7-2600 3.40GHz CPU and 16 GB of main memory. The (wall-clock) time limit was set to 1200 seconds and main memory was limited to 14GB. An extended set of experimental data, as well as source code, benchmarks, and scripts are available at http://fmv.jku.at/cwmulverca. Beside those benchmarks used in our experiments we also include the AIGs we derived for other multipliers used in [28], [24]. More information on the structure of the multipliers used in our experiments can be found in [16], [28], [12] and the README files which come with the experimental data.

In all our experiments the times are listed in seconds (wall-clock time). We measure the time from starting our tool until Mathematica resp. Singular are finished or we reach the time limit (TO), the memory limit (MO), or reach an error state (EE). An error state occurs in Singular when more than 32767 ring variables are allocated. Our results include the time which our tool AigMulToPoly needs to generate the files for the computer algebra system. This time is in the worst case around 3 seconds for 128 bit multipliers. The results also include the time to launch Mathematica resp. Singular.

In Table I we compare our incremental column-wise reduction, outlined in Alg. 1 against the non-incremental approach, where the word-level specification of Def. 2 is reduced against the whole circuit. We apply the non-incremental reduction for column-wise and row-wise ordering. All optimizations (XOR-Rewriting, Common-Rewriting, Vanishing Constraints, Merge, Promote) are enabled. The results in Table I show that in Mathematica and Singular our approach is faster and needs less memory than any non-incremental approach. In the non-incremental experiments, the results between column-wise and row-wise do not really differ. Generally Singular is faster than Mathematica, but it also needs more memory than Mathematica. For multiplier "btor-128" we get an error state.

TABLE I
Incremental (+inc) vs. column- and row-wise non-incr. (-inc).

mult	n	Mathematica			Singular		
		+inc	-inc		+inc	-inc	
			col	row		col	row
btor	16	4	12	12	1	2	2
btor	32	35	531	491	16	53	58
btor	64	409	MO	MO	MO	MO	MO
btor	128	TO	TO	TO	EE	EE	EE
sp-ar-rc	16	7	TO	TO	1	TO	TO
sp-ar-rc	32	67	TO	TO	39	TO	TO
sp-ar-rc	64	841	MO	MO	MO	MO	MO

TABLE II
Effect of turning off optimizations.

mult	n	Mathematica				Singular			
		+inc	-xor	-com	-cs	+inc	-xor	-com	-cs
btor	16	4	TO	1	TO	1	2	1	1
btor	32	35	TO	7	TO	16	64	6	19
btor	64	409	TO	65	TO	MO	MO	MO	MO
btor	128	TO	TO	823	TO	EE	EE	EE	EE
sp-ar-rc	16	7	30	TO	7	1	7	TO	2
sp-ar-rc	32	67	373	TO	64	39	266	TO	34
sp-ar-rc	64	841	TO	TO	805	MO	EE	MO	MO

In the experiments shown in Table II we investigate the effects of turning off optimizations in our column-wise approach and compare these variants to the "+inc" columns of Table I. The results differ for clean and dirty multipliers. For the clean "btor" multipliers turning off Common-Rewriting surprisingly improves the reduction. In this case there are only few gates outside of XORs with only one parent, and splitting remainder computation just increases run-time and space usage. In dirty multipliers, structures like carry trees containing gates with only one parent occur much more frequently. If we turn off Common-Rewriting remainder computation slows down a lot in this case. Turning off XOR-Rewriting influences both clean and dirty multipliers and slows down the reduction (especially in Mathematica), whereas turning off Vanishing Constraints has only a bad effect for clean multipliers in Mathematica, in Singular the results are nearly the same. In summary, the optimizations described in [24] have both positive and negative effects in our experiments, depending on the type of multiplier considered and the computer algebra system used.

TABLE III
DIFFERENCE OF TURNING OFF MERGE AND PROMOTE

mult	n	Mathematica			Singular		
		+inc	-merge	-prom	+inc	-merge	-prom
sp-ar-rc	16	7	8	TO	1	1	TO
sp-ar-rc	32	67	72	TO	39	42	MO
sp-ar-rc	64	841	912	TO	MO	MO	MO

TABLE IV
DIRTY SYNTHESIZED AND MAPPED MULTIPLIERS

mult	n	Mathematica	Singular
abc	8	2	1
abc	16	4	1
abc-resyn3-no-comp	8	351	3
abc-resyn3-no-comp	16	TO	TO
abc-resyn3-comp	8	TO	TO

The experiments shown in Table III compare the effects of turning off our Merge and Promote optimizations on dirty multipliers. In clean multipliers (such as "btor") no gates are merged nor promoted. The running times of Merge enabled or disabled can be considered to be the same. The difference is the size of the carry polynomials, e.g., in sp-ar-rc-8 the carry polynomials have up to 38 monomials with Merge disabled. In our default setting with Merge enabled the biggest carry polynomial contains only 8 monomials and is linear.

In Table IV we also consider synthesized and mapped versions of multipliers. Synthesizing a circuit makes it very hard to verify. When complex mapping is applied it gets even harder and the 8-bit version cannot be verified any more, neither in Mathematica nor Singular confirming [12], [28].

IX. CONCLUSION

We give a simple and precise mathematical formalization of recent successful algebraic approaches to the verification of multiplier circuits, including rigorous proofs of soundness and completeness. We further show how to effectively make use of computer algebra systems. Our main technical contribution is a new incremental column-based verification approach to multipliers, which is an order of magnitude faster than previous row-based approaches relying on reducing a global spec. We further confirm the effectiveness of the algebraic approach and make all data, benchmarks and tools publicly available.

As future work, we want to analyze complexity of previous and our new column-wise approach similar to [7] and [2] and extend our methods to floating-points (following [25]) and other word-level operators. We also want to consider overflow-semantics and negative numbers. An experimental comparison with BMD based techniques should also be performed.

We would like to thank Paul Beame for sharing drafts of [2], Mathias Soeken helping to synthesize AOKI multipliers [16] used in their DATE'16 paper [24], Naofumi Homma sending 128-bit versions of these benchmarks, Maciej Ciesielski explaining the experimental set-up in [12], [28], and finally Deepak Kapur for pointing us to related work [1], [18].

REFERENCES

[1] S. Agnarsson, A. Kandri-Rody, D. Kapur, P. Narendran, and B. Saunders. Complexity of testing whether a polynomial ideal is nontrivial. In *Third MACSYMA User's Conference*, pages 452–458, 1984.

[2] P. Beame and V. Liew. Towards verifying nonlinear integer arithmetic. In *CAV*, volume 10427 of *LNCS*, pages 238–258. Springer, 2017.

[3] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/∼alanmi/abc/. Bitbucket Version 1.01, last change Feb. 27, 2017.

[4] A. Biere. Collection of combinational arithmetic miters submitted to the SAT Competition 2016. In *SAT Competition 2016*, volume B-2016-1 of *Department of Computer Science Series of Publications B*, pages 65–66. Univ. Helsinki, 2016.

[5] A. Biere. Weaknesses of CDCL solvers, August 2016. http://www.fields.utoronto.ca/talks/weaknesses-cdcl-solvers.

[6] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[7] R. Bryant and Y. Chen. Verification of arithmetic circuits using binary moment diagrams. *STTT*, 3(2):137–155, 2001.

[8] B. Buchberger and M. Kauers. Gröbner basis. *Scholarpedia*, 5(10):7763, 2010. http://www.scholarpedia.org/article/Groebner_basis.

[9] J. Chen and Y. Chen. Equivalence checking of integer multipliers. In S. Goto, editor, *ASP-DAC 2001*, pages 169–174, 2001.

[10] Y. Chen and R. Bryant. Verification of arithmetic circuits with binary moment diagrams. In *DAC*, pages 535–541, 1995.

[11] Y. Chen, E. Clarke, P. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao. Verification of all circuits in a floating-point unit using word-level model checking. In *FMCAD*, volume 1166 of *LNCS*, pages 19–33. Springer, 1996.

[12] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi. Verification of gate-level arithmetic circuits by function extraction. In *DAC*, pages 52:1–52:6. ACM, 2015.

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

[14] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 4-1-0. http://www.singular.uni-kl.de, 2016.

[15] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.

[16] N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi. Formal design of arithmetic circuits based on arithmetic description language. *IEICE Transactions*, 89-A(12):3500–3509, 2006.

[17] M. Järvisalo, A. Biere, and M. Heule. Simulating circuit-level simplifications on CNF. *J. Autom. Reasoning*, 49(4):583–619, 2012.

[18] A. Kandri-Rody, D. Kapur, and P. Narendran. An ideal-theoretic approach to work problems and unification problems over finitely presented commutative algebras. In *RTA*, volume 202 of *LNCS*, pages 345–364. Springer, 1985.

[19] I. Koren. *Computer Arithmetic Algorithms*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2001.

[20] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE TCAD*, 21(12):1377–1394, 2002.

[21] J. Lv, P. Kalla, and F. Enescu. Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits. *IEEE TCAD*, 32(9):1409–1420, 2013.

[22] A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0 system description. *JSAT*, 9:53–58, 2014 (published 2015).

[23] T. Pruss, P. Kalla, and F. Enescu. Equivalence verification of large Galois field arithmetic circuits using word-level abstraction via Gröbner bases. In *DAC*, pages 152:1–152:6. ACM, 2014.

[24] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler. Formal verification of integer multipliers by combining Gröbner basis with logic reduction. In *DATE*, pages 1048–1053. IEEE, 2016.

[25] A. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler. Equivalence checking using Gröbner bases. In *FMCAD*, pages 169–176. IEEE, 2016.

[26] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G. Greuel. An algebraic approach for proving data correctness in arithmetic data paths. In *CAV*, volume 5123 of *LNCS*, pages 473–486. Springer, 2008.

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

[28] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski. Formal verification of arithmetic circuits by function extraction. *IEEE TCAD*, 35(12):2131–2142, 2016.