Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model

Maciej Ciesielski*, Tiankai Su*, Atif Yasin*, Cunxi Yu†

***Abstract*—This paper addresses theoretical aspects of arithmetic circuit verification based on algebraic rewriting. Its goal is to advance the understanding of algebraic techniques for arithmetic circuit verification in the context of symbolic computer algebra. The paper offers a new insight into the arithmetic circuit verification problem, by viewing the computation performed by the circuit as the flow of digital data. In the proposed *bit-flow model* the circuit is modeled as a network of logic components satisfying a *bit-flow conservation law*. We prove that the value of the flow of data in the circuit is invariant throughout the circuit and use this to prove soundness and completeness of the rewriting technique, independently from the computer algebra arguments. The efficiency of the method is illustrated with impressive results for large integer multipliers. The verification system and benchmarks are offered in an open source software environment.**

***Index Terms*—Formal Verification, Algebraic Rewriting, Arithmetic Verification.**

I. INTRODUCTION

Despite considerable progress in verification of logic circuits, arithmetic and datapath verification continues to pose a considerable challenge. This may be attributed to the difficulty in efficient modeling of arithmetic designs without resorting to computationally expensive Boolean methods, such as BDDs [1], SAT [2], and SMT [3]. Computer algebra techniques, which are based on polynomial representation of arithmetic circuit implementation, seem to circumvent this problem and offer efficient solutions for analyzing arithmetic circuits and datapaths.

Two flavors of these techniques dominate the field: one, based on Gröbner basis polynomial reduction [4][5][6][7][8][9][10]; and the other, based on algebraic rewriting [11][12]. Although the technique based on algebraic rewriting has been known for several years and proved to be a leading method in arithmetic circuit verification, its theory has not been fully developed. The goal of this paper is to advance the understanding of the algebraic rewriting technique and compare it to an established computer algebra method in order to better explain its merit and efficiency. To this end, we offer a new model, called *bit-flow*, which will be used to prove the merits of the rewriting technique independently from the computer algebra arguments. An open source framework of algebraic rewriting integrated with ABC software [13] is introduced.

The paper is organized as follows: Section II provides the necessary mathematical background of the problem, while Section III reviews prior work in this field. Section IV describes details of the algebraic rewriting scheme and compares it to the Gröbner basis polynomial reduction technique. Section V offers a new insight into the arithmetic circuit verification by introducing the *bit-flow* model. This model provides the basis for the soundness and completeness of the rewriting scheme. Results and conclusions are provided in Sections VI and VII, respectively.

II. THEORETICAL BACKGROUND

The arithmetic circuits considered in this work are circuits whose computation can be expressed as a polynomial in the input variables. These include adders, subtractors, multipliers and fused add-multiply circuits. The circuit is modeled as a network of interconnected bit-level components, each with a finite set of binary inputs and one or more binary outputs. In this work we will focus on gate-level integer arithmetic circuits with single-output logic gates.

Each gate is modeled as a polynomial $f_i \in \mathbb{Z}[X]$, with variables $x_i \in X$ in $\mathbb{Z}_2$. Such polynomials are often referred to as *pseudo-Boolean* polynomials, since they are algebraic expressions with usual multiplication and addition operators over Boolean variables. Formally, a pseudo-Boolean polynomial is an integer-valued function $f : \{0,1\}^n \rightarrow \mathbb{Z}$. The following equations summarize the algebraic representation of the basic Boolean operators:

$$\begin{aligned} \neg a &= 1 - a \\ a \wedge b &= a \cdot b \\ a \vee b &= a + b - a \cdot b \\ a \oplus b &= a + b - 2a \cdot b \end{aligned} \tag{1}$$

By construction, each expression evaluates to a binary value {0,1} and hence correctly models the Boolean function of a logic gate. Models for more complex AOI (And-Or-Invert) gates, used in standard cell technology, are readily obtained from these basic logic expressions. For example, the algebraic model for the logic gate $g = a \vee (b \wedge c)$ can be derived as $g = a + bc - abc$, etc. Similarly, a 3-input OR gate can be represented as $z = a+b+c-ab-ac-bc+abc$, a 3-input XOR gate as $z = a+b+c-2ab-2ac-2bc+4abc$, etc.

To systematically manipulate polynomials, a *term order* "$>$" is imposed on monomials. Let f, g be polynomials, and let $lt(g)$ denote the leading term of polynomial g under such ordering. If a non-zero term t of f is divisible by the leading term of g, we say that f *reduces* to r modulo g, denoted $f \xrightarrow{g} r$, where $r = f - \frac{t}{lt(g)} \cdot g$. Similarly, f can

be reduced w.r.t. a set of polynomials $B = \{f_1, \ldots, f_s\}$, known as polynomial reduction modulo B. It is denoted symbolically as $f \xrightarrow{B}_+ r$, where r is a remainder (also called *normal form*), such that no term in r is divisible by the leading term of any polynomial in B. The sign $+$ refers to the fact that the reduction process is iterative, using polynomials of B one by one.

Let $B = \{f_1, ... f_s\}$ be a set of polynomials representing circuit elements and let R be a polynomial ring, $R = \mathbb{Z}[X]$. Then, $J = \langle f_1, ..., f_s\rangle$ with $f_i \in \mathbb{Z}[X]$, called an *ideal*, is the set of all polynomials generated by f_i, defined as

$$J = \langle f_1, ..., f_s\rangle = h_1 f_1 + ... + h_s f_s : \ h_i \in R \quad (2)$$

The polynomials $f_1, ..., f_s$ are called the bases, or *generators*, of the ideal J. In our case, each generator is a polynomial model of a circuit module, and the set of generators can be viewed as the *implementation* of the circuit. Given an ideal J, the set of all simultaneous solutions to a system of equations $f_1(x_1, ..., x_n) = 0; ..., f_s(x_1, ..., x_n) = 0$ is called *variety*, $V(J)$. From the circuit perspective, a variety contains all signal values of the circuit produced by any set of primary inputs, over all possible input combinations.

The functional specification of the circuit is also defined as a polynomial in $\mathbb{Z}[X]$. For example, the specification of a multiplier circuit $R = A \cdot B$, can then be written as a polynomial $F = R - A \cdot B$ in the input and output variables. Here, A, B and R are symbolic, bit-vector variables, each represented as a polynomial, e.g., $A = \sum_{i=0}^{n-1} 2^i a_i$, etc.

In the terms of computer algebra, the arithmetic circuit verification problem is then formulated as follows [6][7][8][9]: Given a circuit represented by a set of generators (implementation), $B = \{f_1, ..., f_s\}$, and the specification F, the goal of functional verification is to prove that the implementation (B) satisfies the specification (F). This means that the solution to $F = 0$ agrees with $V(J)$, or, equivalently, that F vanishes on $V(J)$[1]. Consequently, this problem has been modeled as an *ideal membership test*, which decides whether polynomial F lies in the ideal J generated by B, i.e., if $F \in J$ [14][6][7].

Given an ideal $J = \langle f_1, ..., f_s\rangle$, to test if $F \in J$, polynomial F is divided sequentially by $f_1, ..., f_s$. The goal is to cancel the leading term(s) of F using one of the leading terms of $f_1, ..., f_s$. Such a reduction results in a polynomial remainder $r = F - \frac{lt(F)}{lt(f_i)} \cdot f_i$, in which the leading term $lt(F)$ has been canceled. If the remainder $r = 0$, the implementation satisfies the specification. However, if $r \neq 0$, such a conclusion cannot be drawn: r can still be in J but is not divisible by polynomials in $B = \{f_1, ..., f_s\}$. That is, the basis $B = \{f_1, ..., f_s\}$ may not be sufficient to reduce $F \longrightarrow 0$, and yet the circuit may be correct. To check if F is reducible to zero for the given ideal J, one must compute a *canonical* set of generators, $G = \{p_1, ..., p_t\}$, called the *Gröbner basis*, such that $\langle p_1, ..., p_t\rangle = \langle f_1, ..., f_s\rangle$. The set G is the Gröbner basis for ideal J iff $\forall F \in J$, $F \xrightarrow{G}_+ 0$ [15]. In short, the Gröbner basis is necessary to unequivocally answer the question whether $F \in J$. A known algorithmic procedure for computing a Gröbner basis is called Buchberger's algorithm [16]. Given some basis $B = \{f_1, ..., f_s\}$, it produces another basis $G = \{p_1, ..., p_t\}$, such that the ideals $\langle p_1, ..., p_t\rangle = \langle f_1, ..., f_t\rangle$ and hence $V(\langle G\rangle) = V(\langle B\rangle)$. Buchberger's algorithm is computationally expensive, as it computes the so-called *S-polynomials* by performing expensive reduction operations on all pairs of polynomials in B. A number of algorithms have been developed for computing a Gröbner basis, including F4 [17], but the process, in general, remains computationally expensive.

[1]Polynomial f is said to vanish on a set V if $\forall a \in V \ f(a) = 0$.

III. Related Work

The work in arithmetic circuit verification was pioneered by [4] and [5], where the concepts from computer algebra and algebraic geometry were applied to model the core verification problem. In [5] an arithmetic circuit is modeled as a network of arithmetic operators, such as half- and full-adders, comparators, and product generators, extracted from the gate-level implementation. These operators are modeled using *arithmetic bit-level* (ABL) expressions, $B = \{B_j\}$. The authors of [5] (and also [7]) show that for an arbitrary combinational circuit, if the terms of the gate equations B are ordered in reverse topological order $\{$outputs$\} > \{$inputs$\}$, then all leading monomials of the polynomials in B are relatively prime. As a result, the corresponding set B already constitutes a Gröbner basis (GB), obviating the computation of the complete canonical basis. The verification problem is solved by reducing the specification modulo B to a *normal form* and testing if it vanishes over $\mathbb{Z}_{2^n}$. In [6], the solution is restricted to binary variables by imposing Boolean constraints, $\langle x^2 - x\rangle$, and the problem is solved over quotient ring $Q = \mathbb{Z}_{2^n}[X]/\langle x^2 - x\rangle$ using a popular computer algebra system, Singular [18]. This approach, however, is limited to circuits composed entirely of half adders and full adders that must first be extracted from the gate-level implementation. In practice, this is the most expensive part of the process, and is not always possible, especially in highly bit-optimized implementations. In [7] the verification problem was similarly formulated as an *ideal membership test* but applied to Galois Field (GF or $\mathbb{F}_{2^q}$) arithmetic circuits. It has been shown that in GF, when the specification F and the ideal J of the circuit implementation are in $\mathbb{F}_{2^q}$, the problem can be reduced to testing if $F \in (J + J_0)$, over a larger ideal $(J + J_0)$ where $J_0 = \langle x^2 - x\rangle$ is an ideal of vanishing polynomials in $\mathbb{F}_2$. Adding J_0 basically restricts the variety V to solutions in $\mathbb{F}_2$, i.e. to $V(J) \cap V(J_0)$ [19]. The polynomials of J_0 are later referred to as *field polynomials*. Similarly to [5], the authors of [7] derive term ordering from the topological structure of the circuit, which renders the set of polynomials B (circuit implementation) a Gröbner basis, thus

obviating the need to perform the expensive GB computation. The method uses a customized, F4-style polynomial reduction using a modified Gaussian elimination algorithm [17] under this term order. A different approach has been proposed in [12], whereby the expensive polynomial reduction has been replaced by a computationally simpler *algebraic rewriting* technique. The method introduces the concept of an *input signature*, a polynomial in the primary inputs, and an *output signature*, a polynomial that encodes the result in terms of the primary outputs. The verification is accomplished by rewriting the output signature, using algebraic expressions of the internal gates, into an input signature, which de facto performs *function extraction*. Several ordering techniques have been described to make this method applicable to large arithmetic circuits, but the method still cannot handle heavily optimized circuits.

A similar approach to arithmetic circuit verification, called *backward construction*, was proposed in 1995 in [20]. It uses *BMDs to reconstruct functional, high level representation from the gate-level structure of arithmetic circuits such as adders and multipliers. Experimental results show that time complexity of the tested circuits is in the order of n^4 for multipliers with n bit operands. There is no clear indication if the *BMD is an efficient datastructure for this problem.

The basic approach of the ideal membership testing and Gröbner basis (GB) reduction has also been used in the works of [8][9], where it was applied to the integer circuits. In [8] the following features have been added to make the reduction more efficient: 1) *Logic reduction* with an AND-XOR vanishing rule, which analyzes the structure of the circuit to identify and remove vanishing monomials that correspond to the product of XOR, AND signals with shared input variables; 2) An *XOR rewriting scheme*, which reduces the model of the circuit to consider only primary inputs, outputs, and fan-out points/XOR gates; and 3) *Common rewriting*, which eliminates the nodes with a single parent. These techniques simplify the task of GB reduction by making the polynomials depend on shared variables, thus increasing the chance for early term cancellation during the rewriting process.

The recent work revisits the techniques from [12] and [8] and provides the proof of correctness for the underlying approaches [9]. It uses a column-wise technique to model and verify basic multiplier structures by computing the Gröbner basis incrementally for each column of the output bit, rather than for the entire circuit. The paper justifies the use of the theory of ideal membership (in principle applicable to $\mathbb{Q}[X]$) to prove properties of integer arithmetic circuits. It points out that, since the leading coefficients of the gate polynomials forming the Gröbner basis are +1 or -1, polynomial reduction never introduces fractional coefficients and their computation remains in $\mathbb{Z}$. This also explains "why dedicated implementations in [12] and [8] can rely on computation in $\mathbb{Z}$ only, while remaining sound and complete" [9]. A follow-up paper, [10], describes an enhancement to this column-wise technique by extracting half- and full-adder constraints to further reduce the size of Gröbner basis to speed up the computation.

In general, the problem of formally verifying complex integer arithmetic circuits (not just multipliers) remains open, and new solutions are being proposed. The remainder of the paper provides a formal analysis of the state-of-the-art approach in this domain based on an algebraic rewriting and introduces a *bit-flow* model to support the proof of the correctness of this approach.

IV. Polynomial Rewriting vs Gröbner Basis Reduction

In this section we analyze the relation between two major techniques used in formal verification of integer arithmetic circuits: algebraic rewriting of [12], and computer algebra-based techniques of [6][8][9].

The function computed by an arithmetic circuit is represented as a *specification* polynomial in the primary input variables, denoted F_{spec}. For example, the specification of an n-bit unsigned integer multiplier, $Z = A \cdot B$ with inputs $A = [a_0, \cdots, a_{n-1}]$ and $B = [b_0, \cdots, b_{n-1}]$, is described by $F_{spec} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2^{i+j} a_i b_j$. The result of the computation, stored in the primary output bits, is also expressed as a polynomial, called *output signature*, S_{out}. Typically, such a polynomial is linear, uniquely determined by the m-bit encoding of the output, provided by the designer. For example, for a signed 2's complement arithmetic circuit with m output bits, $S_{out} = -z_{m-1}2^{m-1} + \sum_{i=0}^{m-2} 2^i z_i$.

The circuit is implemented as a network of logic gates G, each modeled as a polynomial g_i derived from Eqn.(1). The polynomial representing a given gate evaluates to zero for all the input and output combinations satisfied by this gate.

The remainder of this section compares two types of polynomial reduction: 1) based on Gröbner basis (GB) reduction, and 2) based on algebraic rewriting. The results demonstrate that, while both approaches have worst case exponential complexity, the rewriting approach is more efficient. This point will be illustrated with a (non-standard) gate-level implementation of a full adder, shown in Fig. 1.

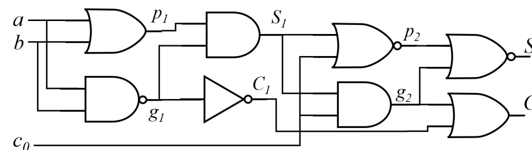

Fig. 1. Gate-level arithmetic circuit (Full Adder)

The following set of polynomials $G = \{f_i\}$ represents the gate-level implementation of the circuit. We refer to this set as G to indicate that it forms a Gröbner basis. The terms of each polynomial are ordered such that the leading

term is the output of the gate, which automatically renders them a Gröbner basis.

$$
\begin{aligned}
f_1 &= p_1 - (-ab + a + b)\\
f_2 &= g_1 - (-ab + 1)\\
f_3 &= S_1 - p_1 g_1\\
f_4 &= C_1 - (-g_1 + 1)\\
f_5 &= p_2 - (S_1 c_0 - S_1 - c_0 + 1)\\
f_6 &= g_2 - S_1 c_0\\
f_7 &= S - (p_2 g_2 - p_2 - g_2 + +1)\\
f_8 &= C - (-C_1 g_2 + C_1 + g_2)\\
f_9 &= (a^2 - a)\\
f_{10} &= (b^2 - b)\\
&\cdots\cdots\\
f_{17} &= (g_2^2 - g_2)
\end{aligned} \tag{3}
$$

Each gate polynomial satisfies the relation $f_i = 0$. The gate polynomials, $f_1, ..., f_8$, have the form $f_i = v_i - tail(f_i)$, where the leading term $lt(f_i) = v_i$ is the output of gate f_i, and $tail(f_i)$ is the logic specification of the gate in terms of its inputs. The leading terms under such ordering are relatively prime, which renders G a Gröbner basis [6][7][9]. This feature is essential for both the GB reduction and the rewriting technique.

The last group of polynomials, $f_9, ..., f_{17}$, represents field polynomials $J_0 =< x^2 - x >$, where x is one of the signals $\{a, b, c_0, p_1, g_1, S_1, C_1, p_2, g_2\}$. They play an important role in the reduction process, which is handled differently in the GB reduction than in the algebraic rewriting approach.

A. *Gröbner Basis Polynomial Reduction*

In this method the reduction of F modulo G is accomplished by successively eliminating terms of F, one by one, by a leading term of some polynomial $f_i \in G$, using Gaussian elimination. The reduction is performed over a Gröbner basis derived from G and field polynomials J_0. From the mathematical point of view, this means that the computation will be performed in the quotient ring, $\mathbb{Z}[X]/\langle x^2 - x\rangle : x \in X$, the set of all variables (signals) of the circuit.

The GB reduction algorithm is given in Algorithm 1. First, the polynomial base G=$\{f_1, ..., f_m\}$ is derived from $\mathcal{N}$ using Equations (1), where m is the number of logic components in $\mathcal{N}$. All the variables in the circuit are ordered in reverse-topological order, from primary outputs to primary inputs, and for each gate polynomial from the gate outputs to its inputs. Furthermore, output signals of gates that depend on common variables (fanins) should be ordered next to each other, as this will maximize the chance for potential term cancellation and minimize the size of intermediate polynomials. For example, consider the reduction of a polynomial $F = 2C + S + ...$ in a circuit containing a half adder composed of an AND gate $C = ab$ and an XOR gate $S = a + b - 2ab$. Since both C and S depend on common variables, a, b, reducing them one immediately after the other will eliminate the product term ab from the polynomial, resulting in $F = a + b + ...$ This is beneficial before continuing with the reduction of the remaining terms of the polynomial.

Considering these two basic ordering rules, one possible term order for the polynomial ring of the circuit in Figure 1 is shown below, where variables in curly brackets can assume any relative order.

$$\{S, C\} > \{p_2, g_2\} > \{S_1, C_1\} > \{p_1, g_1\} > \{a, b, c_0\} \tag{4}$$

The expression F to be reduced is initialized with the difference between the output signature S_{out} and F_{spec}.

Algorithm 1 Gröebner Basis Polynomial Reduction

Input: Specification polynomial F_{spec};and Gate-level netlist $\mathcal{N}$
Output: Remainder Rem

1: Create base G=$\{f_1,...,f_m\}$ of $\mathcal{N}$ using Eq.(1)
2: Generate S_{out} from $\mathcal{N}$
3: Define ring and specify term order
4: Initialize $F \leftarrow S_{out} - F_{spec}$
5: **while** $F \neq 0$ **do**
6:   **if** $\exists f_i \in G : \frac{lt(F)}{lt(f_i)} \neq 0$ **then**
7:   /* there exists f_i such that its leading term is divisible by $lt(F)$ */
8:     $F \leftarrow F - \frac{lt(F)}{lt(f_i)} \cdot f_i$ // polynomial division
9:   **else**
10:   /* no leading term of f_i divides F, move $lt(F)$ to Rem */
11:     $F \leftarrow F - lt(F)$
12:     $Rem \leftarrow Rem + lt(F)$
13:   **end if**
14: Maintain the term order imposed on the ring
15: **end while**
16: **return** Rem

The main part of the GB reduction is given in lines 5-15. The algorithm searches for a polynomial f_i in G such that the leading term of f_i divides the current leading term $lt(F)$ of F. If such a polynomial exists, it will be used to reduce F, as shown in line 8. Otherwise, the $lt(F)$ will be moved to the remainder Rem (lines $11 - 12$). At any point, when new terms (with intermediate variables) are added to polynomial F (line 8), the procedure must maintain the term order imposed on the ring. The reduction process terminates when F becomes empty, either by being reduced or moved to Rem. The zero remainder is the evidence of a correct implementation, as discussed in Section III.

We illustrate the GB reduction process with the example in Fig. 1. The initial polynomial for this circuit is:

$$F = 2C + S - (a + b + c_0)$$

Equation (5) gives a sequence of steps that reduces F with the gate polynomials $f_i \in G$ for the circuit in Figure 1. At each step, F represents the polynomial reduced by the previous reduction step. For brevity, the substitution is shown for a pair of variables at once. For example, $F/(C, S)$ means reducing variables C and S with

$$
\begin{aligned}
&F = 2C + S - (a+b+c_0)\\
1)\ &F/(\ S\ ,\ C\ ) = 2(-C_1g_2 + g_2 + C_1) + (p_2g_2 - p_2 - g_2 + 1) - (a+b+c_0)\\
&\quad = p_2g_2 - p_2 - 2g_2C_1 + g_2 + 2C_1 - (a+b+c_0) + 1\\
2)\ &F/(\ p_2,\ g_2\ ) = (S_1c_0 - S_1 - c_0 + 1)S_1c_0 - (S_1c_0 - S_1 - c_0 + 1) - 2S_1C_1c_0 + S_1c_0 + 2C_1 - (a+b+c_0) + 1\\
&\quad = \mathbf{S_1^2c_0^2} - \mathbf{S_1^2c_0} - \mathbf{S_1c_0^2} + \mathbf{S_1c_0} - 2S_1C_1c_0 + S_1 + 2C_1 - (a+b)\\
3)\ &F/(S_1^2 - S_1) = -2S_1C_1c_0 + S_1 + 2C_1 - (a+b)\\
4)\ &F/(\ S_1,\ C_1) = -2(p_1g_1)(-g_1+1)c_0 + p_1g_1 + 2(-g_1+1) - (a+b)\\
&\quad = -2(-\mathbf{p_1g_1^2} + \mathbf{p_1g_1})\mathbf{c_0} + p_1g_1 - 2g_1 - (a+b) + 2\\
5)\ &F/(\ g_1^2 - g_1) = p_1g_1 - 2g_1 - (a+b) + 2\\
6)\ &F/(\ p_1,\ g_1\ ) = (-ab + a + b)(-ab+1) - 2(-ab+1) - (a+b) + 2\\
&\quad = \mathbf{a^2b^2} - \mathbf{a^2b} - \mathbf{ab^2} + \mathbf{ab}\\
7)\ &F/(\ a^2 -\ a) = 0
\end{aligned}
\tag{5}
$$

polynomials f_8, f_7. The term order imposed on the ring, cf. Eqn. (4), is maintained throughout the entire reduction process.

The effect of field polynomials $J_0 = < x^2 - x >$, responsible for keeping each variable Boolean, can be observed during the steps 2, 4 and 6, shown in bold. The result of the reduction is $Rem = 0$, indicating that the circuit implements the function indicated by the specification, a full adder.

B. Algebraic Rewriting

Algebraic rewriting is the process of transforming the output signature S_{out} into an input signature S_{in} using algebraic models of the internal components (logic gates) of the circuit. The rewriting is done in reverse topological order: from the primary outputs (PO) to the primary inputs (PI); for this reason it is also referred to as a *backward rewriting* [12]. Intermediate expressions obtained during rewriting are also represented as polynomials, referred to as *signatures*, over the variables representing the internal signals of the circuit. By construction, each variable in a given signature (starting with S_{out}) represents an output of some logic gate. The rewriting transformation simply replaces that variable with the algebraic expression of the logic gate. If the variable is part of a monomial involving other variables, the expression is multiplied by the remaining terms and expanded to a disjunctive normal form. This is followed by a standard polynomial simplification by combining terms with same monomials.

The Algebraic Rewriting procedure is summarized in Algorithm 2. First, the polynomial base $G=\{f_1,...,f_m\}$ is derived from $\mathcal{N}$ using Eq.(1), as in the GB reduction. Then, the polynomials in G are sorted in reverse-topological order (lines 1-2). Among several possible topological orders the one that maximizes the number of early cancellations during rewriting is sought. This has an effect of minimizing the size of the intermediate polynomials during rewriting (the "fat belly" effect) [12]. This is accomplished by keeping together the polynomials whose leading terms (gate outputs) depend on common variables, as in the GB reduction. The expression to be rewritten, Sig, is initialized with the given output signature S_{out} of $\mathcal{N}$ (lines 3-4).

Algorithm 2 Algebraic Rewriting

Input: Specification polynomial F_{spec}; and Gate-level netlist $\mathcal{N}$
Output: ($S_{in} == F_{spec}$), or the computed signature S_{in}

```
1: Derive G={f_1,...,f_m} from N using Eq.(1)
2: Sort G to maximize the cancellations // pre-processing
3: Generate S_out from N
4: Initialize Sig ← S_out
5: for f_i in G do
6:     v ← lm(f_i) // leading monomial of f_i is output of a gate
7:     if v ∈ Sig then
8: /* replace v with tail(f_i) in Sig */
9:         Sig ← Sig(v ← tail(f_i))
10:        x ← x^2 // for all x in Sig
11:    end if
12: end for
13: /* upon termination, Sig is composed of PIs only */
14: if Sig == F_spec return True
15: else return S_in = Sig
```

The main part of the rewriting, lines 5-12, iterates over the polynomials $f_i \in G$ and performs the required substitutions. Specifically, all occurrences of $v = lt(f_i)$ in Sig are replaced by $tail(f_i)$, followed by possible expansion of the resulting term. To maintain Boolean value of the variables the degree of each variable in Sig is reduced to 1 (line 10) during rewriting. At the end, the algorithm returns $S_{in} = Sig$ as the derived signature of the circuit. If the terms of polynomials in G are sorted in a reversed topological order, the returned polynomial S_{in} contains only the primary input (PI) variables, so it can be compared with F_{spec}. Detailed proofs of soundness and completeness of the rewriting method are given in Section V.

While the main goal of algebraic rewriting, as described by Algorithm 2, is to determine the arithmetic function implemented by the circuit, it can also be used to verify it against the known specification. This can be simply done by rewriting $F = S_{out} - F_{spec}$ and checking if it produces zero. We will use this rewriting mode in order to compare it against the GB reduction method in Section IV-A.

We illustrate the rewriting process using the example of the gate-level full-adder circuit in Figure 1. The output

$$
\begin{aligned}
F &= 2C + S - (a + b + c_0) \\
1)\ F/(S,\ C\ ) &= 2(C_1 + g_2 - C_1 g_2) + (1 - (p_2 + g_2 - p_2 g_2)) - (a + b + c_0) \\
&= 2C_1 + g_2 - 2C_1 g_2 - p_2 + p_2 g_2 + 1 - (a + b + c_0) \\
2)\ F/(p_2,\ g_2) &= 2C_1 + S_1 c_0 - 2S_1 C_1 c_0 - (1 - (S_1 + c_0 - S_1 c_0)) + (1 - (S_1 + c_0 - S_1 c_0))S_1 c_0 + 1 - (a + b + c_0) \\
&= 2C_1 - 2S_1 C_1 c_0 + S_1 + \mathbf{S_1 c_0} - \mathbf{S_1^2 c_0} - \mathbf{S_1 c_0^2} + \mathbf{S_1^2 c_0^2} - (a + b) \\
&= 2C_1 - 2S_1 C_1 + S_1 - (a + b) \\
3)\ F/(S_1, C_1) &= 2(1 - g_1) - 2(1 - g_1)(p_1 g_1)c_0 + p_1 g_1 - (a + b) \\
&= 2 - 2g_1 - 2(\mathbf{p_1 g_1} - \mathbf{p_1 g_1^2}) + p_1 g_1 - (a + b) \\
&= 2 - 2g_1 + p_1 g_1 - (a + b) \\
4)\ F/(p_1,\ g_1) &= 2 - 2(1 - ab) + (a + b - ab)(1 - ab) - (a + b) \\
&= \mathbf{ab} - \mathbf{a^2 b} - \mathbf{ab^2} + \mathbf{a^2 b^2}\ =\ 0
\end{aligned}
\tag{6}
$$

signature of the circuit is $S_{out} = 2C + S$, determined by the binary encoding of the output, and the specification $F_{spec} = a + b + c_0$. Following the ordering rules described in [12], the best rewriting order which minimizes the size of intermediate polynomials is $\{(S, C), (p_2, g_2), (S_1, C_1), (p_1, g_1)\}$, as in the GB reduction. The signals shown in brackets can be rewritten in any order as they are the ones that depend on common inputs. Equation (6) shows the rewriting steps for the circuit. The terms shown in bold face indicate those that are reduced to zero during polynomial simplification. For brevity, the substitution is shown for each pair of variables applied at once. For example: $F/(C, S)$ means rewriting of F using C and S variables of polynomials f_8, f_7. During the rewriting, two types of simplifications can be observed:

- Simplification of the terms with same monomials; for example, $2g_2 - g_2 = g_2$, in Step 1. This is a common simplification applied in GB reduction as well.
- Lowering the term x^2 to x, since the signal variables are binary. This can be seen in Steps 2, 3, and 4, shown in bold face. For example, in step 2 we have: $S_1 c_0 - S_1^2 c_0 - S_1 c_0^2 + S_1^2 c_0^2 = S_1 c_0 - S_1 c_0 - S_1 c_0 + S_1 c_0 = 0$. Similarly, in step 3: $(p_1 g_1 - p_1 g_1^2) = p_1 g_1 - p_1 g_1 = 0$, etc. This simplification is simpler and can be executed faster than dividing the polynomials by the respective field polynomials $(x^2 - x)$, as it is done in computer algebra approach. This is one of the main reasons for greater efficiency of the algebraic rewriting compared to GB reduction.

Subsequently, the final result reduces $F = S_{out} - F_{spec}$ to zero, indicating that the circuit correctly implements a full adder.

It should be noted that in addition to the two basic simplification rules mentioned above (rewriting the gates with common inputs, and the $x^2 \rightarrow x$ reduction), some more sophisticated simplifications can be applied to the running polynomial Sig during rewriting by analyzing the structure of the gate-level network. For example, recognizing that some signal g is a product of XOR and AND signals with the same fanin inputs will allow it to reduce signal g to zero. This simplification, called an *XOR-AND vanishing rule* has been used by [8], but for clarity of the above illustration, it has not been taken here into account.

C. AIG Rewriting

The algebraic rewriting technique described in the previous section can be further improved by performing rewriting using the functional AIG (Add-Inverter Graph) representation of the circuit instead of its gate level structure. This section provides a brief overview how this is accomplished, with details provided in [21].

AIG (And-Inverter Graph) is a combinational Boolean network composed of two-input AND gates and inverters [13]. Each internal node of the AIG represents a two-input AND function; the graph edges are labeled to indicate a possible inversion of the signal. We use the *cut-enumeration* approach of ABC [13] to detect XOR and Majority (MAJ) functions with a common set of variables; they are essential components of adder trees that are present in most arithmetic circuits in some form [21]. After detecting the XOR and MAJ components of the adder's AIG, rewriting skips over the detected adders, significantly speeding up the rewriting process. Figure 2 illustrates the process for the full adder (FA) circuit from Figure 1. In Figure 2 the groups of nodes (6,7,8) and (9,11,12) correspond to half adders (HA). The functions rooted at nodes 6 and 9 are majority (AND) functions, and those at nodes 12 and 8 are XORs. Subsequently, the functions at node 12 (S) and node 10 (C) are identified as XOR3 and MAJ3, respectively, on the shared inputs, a, b, c_0.

The AIG rewriting of $S_{out} = 2C + S$ over the extracted XOR3 and MAJ3 nodes is trivial, with the nonlinear monomials automatically cancelled as follows:

$$2C + S = 2(ab + ac_0 + bc_0 - 2abc_0)$$

$$+(a + b + c_o - 2ab - 2ac_0 - 2bc_0 + 4abc_0) = a + b + c_o$$

The resulting signature matches the specification, which clearly indicates that the circuit is a full adder. As illustrated with this example, the AIG rewriting requires considerably fewer terms than the standard algebraic rewriting.

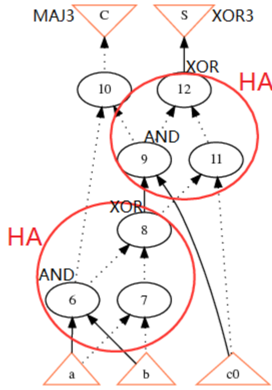

Fig. 2. AIG rewriting of a full adder circuit from Figure 1.

Data structure: AIG rewriting is implemented in ABC with the polynomial data structure, type `Pln_Man_t`. Its main components include: 1) the AIG manager (`Gia_Man`) that represents the input design; and 2) two vector hash tables using type `Hsh_VecMan_t` are used for storing the constants and monomials. The hash tables of monomials include coefficient vectors and monomial vectors. When substitution is applied to the leading term, new monomials will be created and the substituted one will be removed. For example, when $ab+c+bd$ is substituted by $a = b+d$, the monomial ab is removed first, and b and bd are added to `Pln_Man_t`. During the process of adding the new monomials, the program will first check if these monomials already exist in `Pln_Man_t`; in this case only the coefficient of these monomials will be changed accordingly. In this example, two new monomials are generated by the substitution, namely b^2, reduced to b, and bd. Since bd already exists, the coefficient 1 of bd is replaced by 2, resulting in $b+c+2bd$.

D. Comparison between the two Methods

It should be clear from the above discussion that both methods, the GB reduction and algebraic rewriting, are equivalent in the sense that they both perform polynomial reduction. The GB reduction scheme achieves polynomial reduction by division (Gaussian elimination), while algebraic rewriting does it by substituting the gate output variable by the polynomial expression of the gate's function.

While the goal of GB reduction scheme is to reduce $F = S_{out} - F_{spec}$ modulo G to 0, it can also be used to extract the arithmetic function by reducing S_{out} modulo G, and interpret the result as the circuit's functional specification F_{spec}. In the algebraic rewriting scheme, the goal is to rewrite the S_{out} to S_{in}, the expression in the primary inputs, and check if it matches the expected specification F_{spec}. If $S_{in} = F_{spec}$, the circuit is correct; otherwise it is faulty. Alternatively, as illustrated above, algebraic rewriting can be also applied to $F = S_{out} - F_{spec}$, as in the GB approach.

Variable substitution of algebraic rewriting (line 9 of Algorithm 2) seems simpler than the main step of polynomial division of the GB reduction (line 8 of Algorithm 1). On the other hand, it requires additional multiplication of the terms and expansion into a sum of products. Hence, complexity of these steps are comparable. Both methods avoid explicit computation of Gröbner basis, but achieve it by different means. In the GB reduction it is done by setting the variable order in the ring so that all variables are in reverse topological order to make the implementation set G a Gröbner basis. In the algebraic rewriting scheme on the other hand, the polynomials $f_i \in G$ are sorted in reverse topological order to effect the rewriting. As a result, both methods ensure the polynomial base to be a Gröbner basis. However, there are some essential differences between the two methods that affect their efficiency.

- The GB reduction scheme requires the *field polynomials* $J_0 = < x^2 - x >$ to be added to the base G in order to keep the variables Boolean. This increases the size of the Gröbner basis and results in a larger search space in each iteration. Whereas in the rewriting scheme, the reduction by $< x^2 - x >$ is solved in a simpler way by lowering x^2 to x via a simple data structure (line 10 in Algorithm 2).
- In the algebraic rewriting scheme, the gate polynomials $f_i \in G$ are ordered in topological order (line 5 in Algorithm 2) so that each gate polynomial f_i is used exactly once. The selected polynomial is used to perform the rewriting by a simple string substitution and is never needed again. In contrast, in each iteration of the GB reduction one has to search for a polynomial f_i that divides the leading term of F under reduction. While in principle the GB reduction can also work over an ordered list of gate polynomials, this does not apply to the field polynomials $< x^2 - x >$, needed for the reduction. Since the appearance of intermediate signals in nonlinear terms x^k is unpredictable, it is impossible to pre-order the list of field polynomials in GB reduction.

V. The Bit-flow Model

This section offers a new insight into an arithmetic circuit verification problem, in which the computation performed by the circuit is treated as the *flow* of digital data. The goal here is not to introduce any new algorithm, but to suggest an interpretation how the computation propagates in an arithmetic circuit. This interpretation will then provide an argument for soundness and completeness of the algebraic rewriting method, independently from the computer algebra arguments.

The circuit is modeled as an *acyclic network* of logic and/or arithmetic components connected via electrical *signals* or wires. Mathematically, the signals are represented as *variables*, denoted X; they include the internal signals, the primary inputs (PI), and the primary outputs (PO). The

terms *signals* and *variables* will be used interchangeably, depending on the context (structural vs. functional view of the circuit). Each component of the circuit is described by its *characteristic function*, a pseudo-Boolean polynomial function relating the component's inputs to its outputs. The characteristic functions of Boolean logic gates are provided by Equation 1. For example, the characteristic function of an OR gate $z = a \vee b$ is $z = a + b - ab$. Similarly, the characteristic function of a half adder (HA) is $2C + S = a + b$, etc.

The generic term *flow* is intuitively understood as a movement of some physical entity (such as current or fluid) through the network. Here, it is a movement of digital data (voltages evaluated as 0 or 1) whose capacity is measured in bits, where each bit contributes one unit of flow to its value. The flow starts at the primary inputs and propagates towards the primary outputs, distributed internally according to the characteristic functions of the circuit components. For example, a full adder accepts an in-flow of three bits, a, b, c and "distributes" this flow to the outputs according to its characteristic function: $a + b + c = 2C + S$. The coefficient associated with each variable represents its "capacity", the maximum value of the flow that can pass through the corresponding signal. In a half-adder or a full-adder, the weight of each input bit is 1, and the weight of the output bits C and S are 2 and 1, respectively. For a logic gate, the inputs and the output bits have a weight of 1 each.

The idea of using the *flow conservation law* to verify arithmetic circuits has already been proposed in [11]. However, it is applicable only to arithmetic circuits composed of half- and full-adders, where the circuit elements and the specification are modeled as linear expressions. Here, we extend this idea to an arbitrary integer arithmetic circuit which computes an arithmetic function as a polynomial.

The value of the flow in the circuit is captured by the polynomials (signatures) generated during the algebraic rewriting. Equations (5) and (6) are examples of such polynomials. The value of the flow at the primary inputs is represented by the specification polynomial F_{spec}, while the value of the flow at the primary outputs is represented by the output signature S_{out}. The value of the flow at an arbitrary *cut* of the circuit (defined below) is represented by a polynomial in terms of the variables associated with the respective signals of the circuit. It can be computed from the polynomial generated at each step of the algebraic rewriting. We shall show that the $value$ of the flow in an arithmetic circuit represented by such polynomials is *invariant* throughout the circuit.

In principle, the circuit can be composed of arbitrary components, with single-output logic gates as well as multiple-output arithmetic modules, such as half- and full-adders; or any module for which the I/O relationship can be defined as a polynomial. Here we limit our attention to gate-level arithmetic circuits with single-output logic gates. In the remainder of this section, any reference to polynomials S_i, S_{in}, S_{out} or F_{spec} assumes that they are reduced over the field polynomials $< x^2 - x >$, which is implicitly achieved by replacing x^2 with x during the algebraic rewriting (refer to Section IV-B). It should be clear that the value of the flow is not affected by this transformation or by any simplification which removes the terms that evaluate to zero, since it does not change the value of the polynomial.

Consider a polynomial P_i generated at step i of the algebraic rewriting process. It can be observed that the variables X_i that are in the support set of P_i correspond to a *cut* in the circuit. Using network flow terminology, the *cut* is a set of signals that partitions the circuit into two subsets: one containing the gates whose inputs are transitively connected to the primary inputs PI, and the other containing the gates whose outputs are transitively connected to the primary outputs PO. This separation is an inherent property of backward rewriting: starting with the output signature polynomial $P_i = S_{out}$, a variable $x_k \in X_i$ of P_i that represents an output of some gate g_k is replaced by the polynomial in its inputs. From the structural viewpoint, this moves the cut from the gate output to its inputs. From this perspective, the polynomial P_i can also be viewed as the *signature* of the cut C_i, denoted S_i.

Polynomial expressions in Eq. (5) and (6) are examples of cut signatures for the full adder circuit of Figure 1. The input and output signatures, S_{in} and S_{out} defined earlier, are the signatures of the boundary cuts, associated with the primary inputs PI and primary outputs PO, respectively. The following example illustrates the relationship between the polynomial and cut rewriting.

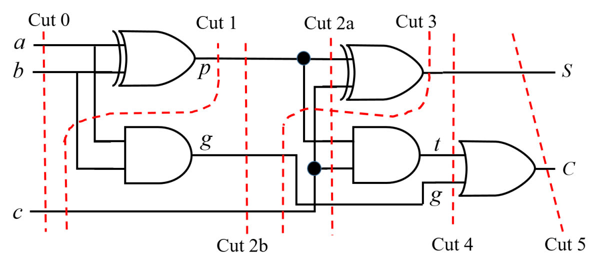

Fig. 3. Cut rewriting in a full-adder circuit.

Example 1: Figure 3 shows a full adder circuit (FA) with a set of cuts. The signatures $\{S_{out}, S_4, S_3, S_2, S_1, S_{in}\}$, associated with cuts $\{Cut_5, ..., Cut_0\}$, are given in Eq. 7. They are obtained by successively rewriting the output signature $S_{out} = 2C + S$ of Cut_5 through the circuit. Specifically, the signature S_{out} is transformed into signature S_4 of Cut_4 by replacing variable C with the expression of the OR gate, $C = g + t - gt$, resulting in the signature $S_4 = 2(g + t - gt) + S$. This signature is then transformed into S_3 by rewriting across the AND gate, $t = cp$, etc., until it reaches the primary inputs. The following

signatures are obtained by successive rewriting of the circuit, in the order consistent with the ordering rules discussed in Section IV. Furthermore, the expression for S_3 is reduced here by applying XOR-AND simplification rule of [8], namely $pg = 0$.

$$\begin{aligned} S_{out} &= 2C + S \\ S_4 &= 2(g + t - gt) + S \\ S_3 &= 2(cp + g - cpg) + S \\ &= 2(cp + g) + S \\ S_2 &= c + p + 2g \\ S_1 &= c + p + 2ab \\ S_{in} &= c + a + b \end{aligned} \tag{7}$$

Note that, in contrast to the network flow model of [11], the signature S_i of some cut C_i is not a linear combination of its signals X_i, but in general a nonlinear polynomial S_i in variables X.

We now introduce the notion of the *flow value*, a measure of the capacity of the bit-flow across a cut. **Definition 1:** The *value* of a cut C_i with signature S_i for a given assignment of variables X_i is an integer value of its signature S_i evaluated at X_i. It is denoted as $V(S_i)(X_i)$.

One should keep in mind that the values of variables X_i of a cut cannot be arbitrary but can assume only those values that can be derived from the bit values of PI. To this effect, we introduce the following definition.

Definition 2: The assignment of variables in X_i is called *legal*, denoted by $[X_i]$, if it is derived from an assignment of the primary inputs, X_{PI}. In this case we say that $[X_i]$ is *compatible* with X_{PI}.

With this we will use the notation $V(S_i)[X_i]$ to denote the value of the cut only for $legal$ assignment of X_i. We can then say that two assignments, $[X_i], [X_j]$, are $compatible$ if they are both derived from the *same* values X_{PI}.

The reason for introducing the concept of legality is that one can only reason about the flow through the cuts for only those values of signals that are actually generated by the circuit. **Example 2:** Table I shows the flow values for the FA circuit in Figure 6 at each cut for all possible PI assignments. These values are obtained by simply substituting given values of $[X_i]$ into the expression of S_i.

An important observation is that, for a given assignment of X_{PI}, the values of all cuts (and of their signatures) are invariant.

Definition 3: Two cuts, C_i, C_j, with signatures S_i, S_j, are *congruent*, denoted $C_i \cong C_j$, if for every pair of compatible assignments, $[X_i], [X_j]$, their values are the same, i.e., $V(S_i)[X_i] = V(S_j)[X_j]$. In this case, we also say that the corresponding signatures are congruent, denoted $S_i \cong S_j$.

TABLE I
FLOW VALUES OF CUTS IN THE CORRECT CIRCUIT.
$S_5 = S_{out} = 2C + S$; $S_0 = S_{in} = a + b + c = F_{spec}$

PIs			Intermediate			POs		Flow value $V(S_i)$ at Cut_i						
c	a	b	p	g	t	C	S	S_5	S_4	S_3	S_2	S_1	S_0	F_{spec}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
0	0	1	1	0	0	0	1	1	1	1	1	1	1	**1**
0	1	0	1	0	0	0	1	1	1	1	1	1	1	**1**
0	1	1	0	1	0	1	0	2	2	2	2	2	2	**2**
1	0	0	0	0	0	0	1	1	1	1	1	1	1	**1**
1	0	1	1	0	1	1	0	2	2	2	2	2	2	**2**
1	1	0	1	0	1	1	0	2	2	2	2	2	2	**2**
1	1	1	0	1	0	1	1	3	3	3	3	3	3	**3**

Theorem 1: Given a pair of cuts C_i, C_j, such that C_i is transformed into C_j or, equivalently, S_i rewritten into S_j by algebraic rewriting, the two cuts are congruent. That is $S_i \longrightarrow S_j \implies S_i \cong S_j$.

Proof. A cut $C_i(X_i)$ can be transformed into another cut $C_j(X_j)$ by a series of algebraic rewriting transformations over logic gates, each described by some polynomial $g = v - tail(g)$. During rewriting, every occurrence of variable v in the source cut (initially C_i) is replaced by $tail(g)$ in the target cut (finally C_j). Since polynomial g satisfies the relation $g = v - tail(g) = 0$, provided by Eq. (1), then $v = tail(g)$. Consequently, $V(S_i) = V(S_j)$ for all values of variables v and those in $tail(g)$ that satisfy this relation. Hence, $V(S_i)[X_i] = V(S_j)[X_j]$ for all compatible assignments $[X_i], [X_j]$, and thus by Definition 6 they are congruent, $S_i \cong S_j$. □

Example 3: Theorem 1 states an important property of bit-flow conservation across the cuts in an arithmetic circuit. Table I gives the values of individual cuts for the full-adder circuit in Figure 3. As we can see, the signature value of each cut in the original (correct) circuit, including the inputs and output signatures are the same for all primary input assignments.

Notice that two cuts may be congruent even if one cannot be obtained from the other by rewriting. For example, in Figure 3, $Cut_3 = \{S, c, p, g\}$ and cut $\{p, c, t, g\}$ (crossing each other, not shown in the figure) cannot be derived from each other since there are no gates that can transform one into another; yet, they are also congruent since each can be derived by a rewriting of S_{out}, albeit through a different set of gates. To that effect, we have the following Corollary:

Corollary 1: All cuts in the circuit are mutually congruent. In particular, $S_{out} \cong S_{in}$.

Proof. By Theorem 1, any cut C_i in the circuit is congruent with the cut at the primary outputs, PO, because it can be obtained by backward rewriting from PO. Any other cut, C_j, is also congruent to PO. That is, by definition of congruence, $V(S_i)[X_i] = V(S_{PO})[X_{PO}]$ and $V(S_j)[X_j] = V(S_{PO})[X_{PO}]$, and hence $S_i \cong S_j$, for any cuts C_i, C_j, including S_{in} and S_{out}. As a result, all the cuts are congruent and form an equivalence class. □

Corollary 1 basically states that the value of the flow measured at *any* cut in the circuit is constant throughout the circuit.

We now need to discuss how to distinguish a circuit that is functionally correct from the circuit that is faulty. The circuit is said to be functionally correct if its implementation satisfies the specification; or, equivalently, that the values computed by the circuit are the same as those provided by the specification for all possible input assignments. Using the terminology of algebraic rewriting we can formalize this definition as follows:

Definition 4: The circuit is *functionally correct*, if for each primary input assignment, X_{PI}, the result encoded in the primary outputs X_{PO} satisfies the condition $V(S_{out})[X_{PO}] = V(F_{spec})[X_{PI}]$.

The following theorem specifies the sufficient and necessary condition for the functional correctness of a circuit.

Theorem 2: The circuit is functionally correct if and only if the *input signature*, S_{in}, computed by algebraic rewriting of the output signature, S_{out}, is the same as the *functional specification*, i.e., if $S_{in} = F_{spec}$.

Proof. The *if* part (soundness): let $S_{in} = F_{spec}$, which implies that $V(S_{in}) = V(F_{spec})$ for all possible primary input assignments, X_{PI}. Since, by Corollary 1, $S_{in} \cong S_{out}$, i.e., $V(S_{in}) = V(S_{out})$, we have $V(S_{out}) = V(F_{spec})$ for all possible values of X_{PI}. That is, the circuit is functionally correct.

The *only if* part (completeness): Let the circuit be functionally correct, i.e., $V(S_{out}) = V(F_{spec})$ for all values of X_{PI}. Since $S_{out} \cong S_{in}$, we have $V(S_{in}) = V(F_{spec})$ for all the assignment of inputs X_{PI}. This in turn implies that $S_{in} = F_{spec}$. Furthermore, the rewriting procedure always terminates: the circuit as a DAG has no loops and the number of rewriting steps is equal to the number of gates. Hence, the method is also complete.

It should be emphasized that the above argument is only valid for pseudo-Boolean polynomials, reduced over field polynomials J_0. It is known that such polynomials have unique polynomial representation, so that two polynomials will evaluate to the same value only if they are the same. □

Example 4: To illustrate the case of a faulty circuit, where $S_{in} \neq F_{spec}$, consider again the full adder example in Figure 3 in which the AND gate $g = ab$ has been replaced with an OR gate, $g = a+b-ab$. This causes the signatures of the cuts to change, as follows (note that in this circuit the AND-XOR simplification $pg = 0$ does not apply):

$$\begin{aligned} S_{out} &= 2C + S \\ S_4 &= 2(g + t - gt) + S \\ S_3 &= 2(cp + g - cpg) + S \\ S_2 &= c + p + 2g - 2cpg \\ S_1 &= c + p + 2(a + b - ab) - 2cp(a + b - ab) \\ S_{in} &= c + 3(a + b) - 4ab - 2c(a + b - 2ab) \end{aligned} \tag{8}$$

The input signature obtained by this rewriting is now: $S_{in} = c + 3(a + b) - 4ab - 2c(a + b - 2ab)$, which does not match the circuit specification, $F_{spec} = a + b + c$. The flow values for each cut, for each assignment X_{PI}, are shown in Table II. The table confirms that all the cuts $\{S_5, S_4, S_3, S_2, S_1, S_0\}$ are congruent; and the flow value at any of the cuts, according to Theorem 1, is constant for any PI assignment. However, the flow value for some assignments of X_{PI} is different than in the correct circuit (shown in the column F_{spec}), proving that the circuit is faulty.

TABLE II
FLOW VALUES IN FAULTY CIRCUIT (GATE AND OF g REPLACED BY OR); $S_5 = S_{out} = 2C + S$; $S_0 = Sin \neq F_{spec}$

PIs			Intermediate			POs		Flow value $V(S_i)$ at Cut_i						
c	a	b	p	g	t	C	S	S_5	S_4	S_3	S_2	S_1	S_0	F_{spec}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
0	0	1	1	1	0	1	1	3	3	3	3	3	3	**1**
0	1	0	1	1	0	1	1	3	3	3	3	3	3	**1**
0	1	1	0	1	0	1	0	2	2	2	2	2	2	**2**
1	0	0	0	0	0	0	1	1	1	1	1	1	1	**1**
1	0	1	1	1	1	1	0	2	2	2	2	2	2	**2**
1	1	0	1	1	1	1	0	2	2	2	2	2	2	**2**
1	1	1	0	1	0	1	1	3	3	3	3	3	3	**3**

In summary, in the circuit that computes a polynomial, the value of the flow from PI to PO is *constant* throughout the entire circuit. In the functionally correct circuit the value of the flow equals that of F_{spec}; in a faulty circuit the flow value is different than that of F_{spec}, while all the cuts remain congruent.

If the circuit is correct, S_{in} will match the specification, F_{spec}; otherwise, the algorithm will report the circuit as faulty and will return the computed signature S_{in}.

VI. RESULTS

Our algebraic rewriting algorithm has been implemented in C and integrated with the ABC tool [13], where it is performed over the AIG datastructure. We developed an **open source** framework of Algebraic RewriTing (ARTi) system for arithmetic circuit verification using ABC [13] as back-end[2], for open access and reproducibility.

The experiments were conducted on benchmarks released in [9][10][3]. For fair comparison, we recompiled their C code on our platform and evaluated it with Singular v4.1.1 [18]. The experiments were conducted on a PC with Intel(R) Xeon CPU E5-2420 2.20 GHz x24 with 1

[2]Source and demos: https://github.com/ycunxi/abc
[3]http://fmv.jku.at/algeq/

TB memory. The memory out (MO) limit is 100 GB and timeout (TO) limit is 3600 seconds. Singular reports error state (ES) if the circuit contains more than 32,767 ring variables.

The verification results for multipliers without synthesis and technology mapping are included in Table III, and those with mapping are given in Table IV . The results in column ARTi are generated for three types of circuits, *btor, sp-ar-rc*, and multipliers generated by *abc*, using the following sets of commands for:

a) *read btorXX.aig; &get; &polyn -v;*

b) *read sp-ar-rcXX.aig; &get; &atree; &polyn -v;*

c) *gen -N XXX -m abcXXX.blif; &get; &polyn -v;*

The command &$polyn$ includes various rewriting options, such as using the structural gate-level netlist, AIG datastructure, signed or unsigned circuits, verbosity level (-v), automatic vs manual specification of the output signature, and more. Details are available from the ABC tool, command &$polyn - help$. Extraction of the adder tree is invoked by the command &$atree$.

TABLE III
CPU VERIFICATION TIME (IN SECONDS) FOR MULTIPLIERS PRIOR TO SYNTHESIS. ES = ERROR STATE REPORTED BY SINGULAR.

Design	ARTi	[9]	[10]
btor-16	0.01	0.5	0.01
btor-32	0.02	11.7	0.3
btor-64	0.1	725	4.0
btor-128	0.5	ES	ES
sp-ar-rc16	0.01	1.1	0.01
sp-ar-rc32	0.1	35.5	0.3
sp-ar-rc64	0.4	1312	4.6
sp-ar-rc128	1.6	ES	ES
abc-256	0.7	ES	ES
abc-512	3.7	ES	ES

Table IV shows the verification results for multipliers mapped onto standard cells with three different libraries, including simple two-input gates and *industrial libraries* of 14 nm and 7 nm nodes. The table also compares the results with the open source tools of [9][10]. The first group of four designs in the table are the synthesized circuits without technology mapping. The three circuits in the second group are synthesized and mapped onto a simple library of two-input gates. The last group of four circuits contains designs that were synthesized and mapped onto industrial libraries. For these circuits we executed several iterations of *dch* and *strash* commands before applying ARTi to eliminate extra logic introduced for meeting timing constraints. As can be seen from the tables, our algebraic rewriting is significantly more efficient than those using computer algebra, GB-reduction based approach.

We were unable to directly compare our results with those of [20] for the lack of benchmarks and access to their code. This paper, dating 1995, reports that a 64-bit gate-level multiplier can be verified by reconstructing it into a *BMD in 3-6 hours on a SPARCstation 10/51, which is an impressive result for the time. Our attempts to represent large arithmetic circuits with canonical representations such as *BMD or TED were not successful.

TABLE IV
CPU VERIFICATION TIME (IN SECONDS) OF SYNTHESIZED AND TECHNOLOGY MAPPED MULTIPLIERS USING DIFFERENT LIBRARIES. #GT = NUMBER OF GATE TYPES. FI≥5 = NUMBER OF GATES WITH FANIN≥5.

Designs	ARTi	#GT	FI≥5	[9]	[10]
btor64-resyn3-nomap	0.1	-	-	711	4.2
abc64-resyn3-nomap	0.1	-	-	801	4.0
btor128-resyn3-nomap	0.3	-	-	ES	ES
abc128-resyn3-nomap	0.1	-	-	ES	ES
btor64-resyn3-map-simple	0.3	7	0	1073	418
abc64-resyn3-map-simple	0.1	7	0	1071	415
abc128-resyn3-map-simple	1.8	7	0	ES	ES
abc64-resyn3-map-14nm	29	15	17	TO	TO
abc64-resyn3-map-7nm	MO	24	9,791	TO	TO
abc128-resyn3-map-14nm	400	15	1,008	ES	ES
abc128-resyn3-map-7nm	MO	23	26,600	ES	ES

VII. CONCLUSIONS

The paper addresses theoretical aspects of arithmetic circuit verification based on algebraic rewriting in the context of symbolic computer algebra. It provides a detailed comparison between both methods and explanation why the rewriting scheme is more efficient than the GB reduction scheme. The bit-flow model is introduced to formally prove the rewriting approach.

Two modes of algebraic rewriting are possible: 1) Verification against the known specification; and 2) Extracting the specification from the circuit structure. If the specification of the circuit is known, one needs to compare the computed input signature with this specification. While this can be done using canonical polynomial representations, such as TED or BMD, this comparison can be avoided altogether by rewriting the *difference* between the output and input signature, $S_{out} - F_{spec}$ instead of S_{out}. The result of such a rewriting should be zero for a correct circuit. A non-zero result is an indication of a bug. In the case when specification is not known, the computed input signature provides the function of the circuit (buggy or not). In the case of a buggy circuit, the size of intermediate polynomials during rewriting may become prohibitively large, sometimes even preventing the computation from completing. This by itself can be used as a warning signal that the circuit is probably faulty. In general, concluding that the circuit is incorrect and identifying a bug is a challenging problem. Several attempts have been made to identify the bug(s), either by comparing the result of backward and forward rewriting [22] or by analyzing the difference between the computed input signature and the given specification [23]. With a notable exception of finite field (GF) arithmetic circuits [24][25], the debugging of arithmetic circuits remains an open problem.

While the bit-flow verification model presented in this paper does not offer any particular algorithmic method per se, it gives an interesting interpretation of the computation

performed by the circuit. It also provides arguments for the proof of the correctness of the rewriting-based verification: with the bit-flow model, algebraic algebraic rewriting is proved to be sound and complete. The method can be used to verify an arbitrary arithmetic circuit, on an arbitrary level of abstraction (not only gate-level), as long as its functional specification F_{spec} and an output encoding S_{out} can be expressed as a polynomial. An open source framework with various backward rewriting options are released publicly.

ACKNOWLEDGMENTS

This paper was supported by a grant from the National Science Foundation, Award No. CCF-1617708. We are indebted to Prof. Priyank Kalla, University of Utah, for explaining mathematical concepts of computer algebra; and to Prof. Hans Schönemann and Dr. Christian Eder, Univeristy of Kaiserslautern, for their help with using the Singular software.

REFERENCES

[1] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[2] M. Ganai and A. Gupta, *SAT-based scalable formal verification solutions*. Springer, 2007.

[3] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, 2015.

[4] N. Shekhar, P. Kalla, and F. Enescu, "Equivalence Verification of Polynomial Data-Paths Using Ideal Membership Testing," *TCAD*, vol. 26, no. 7, pp. 1320–1330, July 2007.

[5] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G.-M. Greuel, "An Algebraic Approach for Proving Data Correctness in Arithmetic Data Paths," *CAV*, pp. 473–486, July 2008.

[6] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, A. Dreyer, F. Seelisch, and G. Greuel, "Stable: A new qf-bv smt solver for hard verification problems combining boolean reasoning with computer algebra," in *DATE*, 2011, pp. 155–160.

[7] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner Basis Reductions for Formal Verification of Galois Field Arithmatic Circuits," *TCAD*, vol. 32, no. 9, pp. 1409–1420, September 2013.

[8] 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'16*, 2016, pp. 1–6.

[9] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD'17*, 2017.

[10] ——, "Improving and extending the algebraic approach for verifying gate-level multipliers," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1556–1561.

[11] M. Ciesielski and A. R. W. Brown, "Arithmetic Bit-level Verification using Network Flow Model," in *Haifa Verification Conference, HVC'13*. Springer, LNCS 8244, Nov. 2013, pp. 327–343.

[12] C. Yu, W. Brown, D. Liu, A. Rossi, and M. J. Ciesielski, "Formal verification of arithmetic circuits using function extraction," *TCAD*, vol. 35, no. 12, pp. 2131–2142, 2016.

[13] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *Proc. Intl. Conf. on Computer-Aided Verification*, 2010, pp. 24–40.

[14] S. Gao, "Counting zeros over finite fields with gröbner bases," *Master's thesis, Carnegie Mellon University*, 2009.

[15] W. Adams and P. Loustanau, *An Introduction to Gröbner Bases*. American Mathematical Society, 1994.

[16] B. Buchberger, "Ein algorithmus zum auffinden der basiselemente des restklassenringes nach einem nulldimensionalen polynomideal," Ph.D. dissertation, Univ. Innsbruck, 1965.

[17] J.-C. Faugere, "A New Efficient Algorithm for Computing Gröbner Bases (F4)," *Journal of Pure and Applied Algebra*, vol. 139, no. 1–3, pp. 61 – 88, 1999.

[18] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann, "SINGULAR 3-1-6 A Computer Algebra System for Polynomial Computations," Tech. Rep., 2012, http://www.singular.uni-kl.de.

[19] D. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms*. Springer, 1997.

[20] K. Hamaguchi, A. Morita, and S. Yajima, "Efficient construction of Binary Moment Diagrams for verifying arithmetic circuits," in *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, Nov 1995, pp. 78–82.

[21] C. Yu, M. J. Ciesielski, and A. Mishchenko, "Fast algebraic rewriting based on and-inverter graphs," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1907–1911, 2018.

[22] S. Ghandali, C. Yu, D. Liu, W. Brown, and M. Ciesielski, "Logic debugging of arithmetic circuits," in *ISVLSI'15*, July 2015, pp. 113–118.

[23] F. Farahmandi and P. Mishra, "Automated test generation for debugging multiple bugs in arithmetic circuits," *IEEE Transactions on Computers*, 2018.

[24] T. Su, A. Yasin, C. Yu, and M. J. Ciesielski, "Computer algebraic approach to verification and debugging of galois field multipliers," in *IEEE International Symposium on Circuits and Systems, ISCAS 2018, 27-30 May 2018, Florence, Italy*, 2018, pp. 1–5.

[25] V. Rao, U. Gupta, I. Ilioaea, A. Srinath, P. Kalla, and F. Enescu, "Post-Verification Debugging and Rectification of Finite Field Arithmetic Circuits using Computer Algebra Techniques," *FMCAD'18*.