Equivalence Checking using Gröbner Bases

Amr Sayed-Ahmed[1] Daniel Große[1,2] Mathias Soeken[3] Rolf Drechsler[1,2]
[1]Faculty of Mathematics and Computer Science, University of Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany [3]Integrated Systems Laboratory (LSI), EPFL, Switzerland
{asahmed,grosse,drechsle}@informatik.uni-bremen.de mathias.soeken@epfl.ch

***Abstract*—Motivated by the recent success of the algebraic computation technique in formal verification of large and optimized gate-level multipliers, this paper proposes algebraic equivalence checking for handling circuits that contain both complex arithmetic components as well as control logic. These circuits pose major challenges for existing proof techniques. The basic idea of Algebraic Combinational Equivalence Checking (ACEC) is to model the two compared circuits in form of Gröbner bases and combine them into a single algebraic model. It generates bit and word relationship candidates between the internal variables of the two circuits and tests their membership in the combined model. Since the membership testing does not scale for the described setting, we propose reverse engineering to extract arithmetic components and to abstract them to canonical representations. Further we propose arithmetic sweeping which utilizes the abstracted components to find and prove internal equivalences between both circuits.**

We demonstrate the applicability of ACEC for checking the equivalence of a floating point multiplier (including full IEEE-754 rounding scheme) against several optimized and diversified implementations.

***Index Terms*—Formal Verification, Equivalence Checking, Gröbner Bases, Reverse Engineering, Floating-Point Multiplier.**

I. Introduction

Arithmetic circuits are typically difficult instances for classical Boolean reasoning approaches that are based on, e.g., *Binary Decision Diagrams* (BDDs) or *Boolean Satisfiability* (SAT), as they suffer from exponential worst-case complexity. Boolean reasoning based on Gröbner bases (available with algebraic computation packages) offers a robust mechanism that verifies arithmetic circuits at gate-level (see, e.g., [1], [2]) and their power has recently been demonstrated in formally verifying large and optimized gate-level multipliers [3]. However, circuits that also contain control logic pose a major difficulty for algebraic computation based reasoning techniques and no satisfactory solution has yet been presented. In this paper, we show techniques that allow to reason over circuits which combine data-path and control logic using symbolic computation reasoning. To the best of our knowledge, this is the first full automated technique that formally verifies binary floating-point circuits without any kind of case splitting or other manual effort.

So far, verification using algebraic computation models the circuit under verification as polynomials $G = \{g_1, \dots, g_k\}$ and tests the membership of the specification polynomial p_{spec} in G. The polynomials in G contain *internal variables* for all gates in the circuit, whereas p_{spec} is expressed only in terms of the primary inputs and primary outputs (n input bits and m output bits in total). p_{spec} can be viewed as a map over the finite integer space, i.e., $p_{\text{spec}} : \mathbb{Z}_{2^n} \to \mathbb{Z}_{2^m}$. Membership testing is performed by reducing (dividing) p_{spec} wrt. G. If this reduction completes with no remainder, the circuit fulfills the specification. During the reduction, it is possible that the intermediate polynomials blow up which can be eluded by applying intermediate rewriting strategies (see, e.g., [3]).

Motivated by the fundamental problem that not every circuit specification p_{spec} can be represented in a canonical and abstract form over $\mathbb{Z}_{2^n}$, we are interested in *equivalence checking*, i.e., we want to prove the functional equivalence of two circuits in the absence of a specification. This can be done as follows: Assume the two circuits checked for equivalence represent the functions $f_1(x_1, \dots, x_n) = (y_1, \dots, y_m)$ and $f_2(x_1, \dots, x_n) = (z_1, \dots, z_m)$ and are given as two sets of polynomials G_1 and G_2. Then we divide each polynomial $z_j - y_j$ for $1 \le j \le m$—which formulates the equivalence of each output bit—by polynomials from the combined model $G = G_1 \cup G_2$. This naïve method does not scale since during the reduction the internal variables in the polynomials in G cause for a tremendous overhead which can only be resolved when the primary input variables x_i appear in the polynomials.

This problem can be circumvented if one knows internal equivalences in the two circuits which allows to put internal variables into relation. Conceptually, this is similar to SAT sweeping and as a consequence G is simplified. This ultimately avoids a blow-up of the polynomials during reduction. The difficulty is finding internal equivalences. To solve this problem we propose reverse engineering techniques: First, expected arithmetic word-level components such as multipliers and adders are detected in the circuit using structural signatures. Then, the proposed arithmetic sweeping uses the I/O boundaries of detected word-level components to prove internal equivalences and to prevent division blow-ups.

To further reduce verification runtime during the divisions we propose decomposition and a general reduction rule that allow more compact representations and semi-canonical representations for different implementations of the same function.

The result is a new *Algebraic Combinational Equivalence Checking* (ACEC) technique which is based on Gröbner bases. In contrast to classical combinational equivalence checking [4], [5], it can check the equivalence of two circuits which contain different architectures of arithmetic units, e.g multipliers and adders, as well as control logic parts. Our experimental evaluation demonstrates the applicability of our algebraic equivalence checking approach on several optimized floating-point multipliers which cannot be verified by other proof techniques.

II. Preliminaries

Using concepts from algebraic geometry and symbolic computation, we model the given combinational circuits with

This work was supported in part by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1, by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative, by H2020-ERC-2014-ADG 669354 CyberCare, and by the German Academic Exchange Service (DAAD). Also, we thank OneSpin Solutions for making their EC tool available to us.

a set of multivariate polynomials based on Gröbner bases and we formulate the equivalence checking problem as testing the membership of some relationships between these circuits using the ideal membership concept. In the following, we define common notations of these algebraic concepts based on [6]. Then, we present the Gröbner bases modeling and the ideal membership testing algorithm.

A. Notation and Definitions

The ring of integers modulo 2 ($\mathbb{Z}_2$) is called a *Boolean ring*. As is shown in [7], [8], theory of Gröbner bases can be applied on Boolean rings, it is referred as a *Boolean Gröbner bases*. For a Boolean polynomial ring $\mathbb{Z}_2(x_1, \dots, x_n) = \mathbb{Z}_2[x_1, \dots, x_n]/\langle -x_1^2 + x_1, \cdots, -x_n^2 + x_n\rangle$ of n Boolean variables, the polynomials $\langle -x_i^2 + x_i\rangle$ are added to the polynomial ring $\mathbb{Z}_2[x_1, \dots, x_n]$ to keep the variables x_i in the Boolean domain. A *monomial* $M = x_1^{\alpha_1} \cdots x_n^{\alpha_n}$ is the power product over the variables $x_1, \dots, x_n$. As for Boolean variables $x_i^2 = x_i$, the powers α_i are always equal to one. A *polynomial* $p = c_1M_1 + \cdots + c_tM_t$ is a finite sum of terms, where each term is the product of an coefficient c_i and a monomial M_i. The monomials of a polynomial are ordered according to a *monomial ordering* '$>$', such that $M_1 > \cdots > M_t$, the *leading term* of the polynomial is $\mathrm{lt}(p) = c_1M_1$, the *leading monomial* is $\mathrm{lm}(p) = M_1$, and the *leading coefficient* is $\mathrm{lc}(p) = c_1$. We denote $\mathrm{tail}(p) = p - \mathrm{lt}(p) = c_2M_2 + \cdots + c_tM_t$.

In this work, the monomial order follows the reverse topological order of the variables of the modeled circuit and the coefficient $c_i \in \mathbb{Z}$ for all $i \neq 1$, where the leading coefficient $\mathrm{lc}(p) = c_1 \in \{-1, 1\}$. The coefficients c_i are not limited to $\{0, 1\}$ as in Galios Field $\mathbb{GF}_2$, they could be arbitrary as shown in [7] or integers as in this work.

For a set of polynomials $P = \{p_1, \dots, p_s\} \in \mathbb{Z}_2(x_1, \dots, x_n)$, an *affine variety* $V(p_1, \dots, p_s)$ is the set of all solutions of the polynomial equations $p_1(x_1, \dots, x_n) = \cdots = p_s(x_1, \dots, x_n) = 0$. The affine variety depends not just on the given set of polynomials, but rather on the ideal generated by the polynomials. An *ideal* $I = \langle P\rangle = \{\sum_{i=1}^{s} h_i \cdot p_i : h_i \in \mathbb{Z}_2[x_1, \dots, x_n]\}$ is generated by this set of polynomials P, and we call P the bases (generators) of the ideal I. The ideal I may have many other bases. The bases are different representations of the set of polynomials P. One of these bases is called *Gröbner bases* $G = \{g_1, \dots, g_{\hat{s}}\}$, for which $V(G) = V(I)$. Gröbner bases reveal the properties of the ideal that allow to solve the ideal membership testing problem in an algorithmic fashion.

Definition 1: A *polynomial division* of two polynomials p and g denoted as $p \xrightarrow{g}_+ r$ is performed as $r = p - \frac{cM}{\mathrm{lt}(g)}g$. If a non-zero term cM of p is divisible by the leading term of g, then p reduces to r modulo g. Similarly, p can be reduced (divided) wrt. a set of polynomials P to obtain a remainder r, denoted $p \xrightarrow{P}_+ r$, such that no term in r is divisible by the leading term of any polynomial in P.

Definition 2: A polynomial reduction method named *S-polynomial* of polynomials p and g in a polynomial set P, is the combination $\mathrm{Spoly}(p, g) = \frac{L}{\mathrm{lt}(p)}p - \frac{L}{\mathrm{lt}(g)}g$, where L is the least common multiple $\mathrm{LCM}(\mathrm{lm}(p), \mathrm{lm}(g))$.

To compute the Gröbner bases $G = \{g_1, ..., g_{\hat{s}}\}$ for an ideal $I\langle p_1, \dots, p_s\rangle$, *Buchberger's algorithm* constructs G in a finite the number of steps by applying $\mathrm{Spoly}(p, g) \xrightarrow{G}_+ r$ in every step. Gröbner bases are computed if all $\mathrm{Spoly}(p, g) \xrightarrow{G}_+ 0$.

Lemma 1: Given a finite set $G \in \mathbb{Z}_2(x_1, \dots, x_n)$, suppose that we have $p, g \in G$ such that $\mathrm{LCM}(\mathrm{lm}(p), \mathrm{lm}(g)) = \mathrm{lm}(p) \cdot \mathrm{lm}(g)$. In other words, the leading monomials of p and g are relatively prime. Then $\mathrm{Spoly}(p, g) \xrightarrow{G}_+ 0$ [6].

According to Lemma 1, a given polynomial set is a Gröbner basis, if the leading monomials of all polynomials in the set are relatively prime. By combining this lemma with the affine variety concept of an ideal, we define the Gröbner bases of an ideal as follows:

Definition 3: A finite subset $G = \{g_1, \dots, g_{\hat{s}}\}$ wrt. a monomial order of an ideal I is said to be a Gröbner basis of I if $V(G) = V(I)$ and all leading monomials in G are relatively prime.

A given ideal may have different Gröbner bases, where one basis can be reduced to other bases by eliminating (substituting) some of ideal variables based on the *Elimination Theorem* [6], in the following, this process is named model rewriting. These bases can be reduced again to a canonical representation of the ideal that is called *reduced Gröbner basis*.

Definition 4: A reduced Gröbner basis for a polynomial ideal I is a Gröbner basis G for I, such that for all $g_i \in G$, no term in g_i is divisible by the leading term $\mathrm{lt}(g_j)$ for all $i \neq j$.

Lemma 2: Let $I \neq 0$ be a polynomial ideal. Then, for a given monomial ordering $>$, I has a unique reduced Gröbner basis [6].

We utilize the uniqueness property of the reduced Gröbner basis for canonical polynomial abstraction in Section IV.

The *Ideal Membership Testing* (IMT) decides whether a given polynomial p lies in the Gröbner basis ideal $G = \{g_1, \dots, g_{\hat{s}}\}$. It applies a division algorithm to check that the remainder r on dividing p by G is equal to zero. The division is denoted $p \xrightarrow{G}_+ r$.

B. Modeling a Circuit as Gröbner Basis

Logic gates are modeled by polynomials and signals as Boolean variables. The polynomials of basic Boolean gates are

$$\begin{aligned}
z = \neg a &\implies g := -z + 1 - a \\
z = a \wedge b &\implies g := -z + ab \\
z = a \vee b &\implies g := -z + a + b - ab \\
z = a \oplus b &\implies g := -z + a + b - 2ab.
\end{aligned}$$

Each logic gate is modeled in a way that the gate output variable z is described in terms of the gate input variables a, b. The polynomial $x^2 - x$ is added to the model for each variable to enforce the Boolean domain. In practice, the ideal polynomials $\langle -x^2 + x\rangle$ are replaced by reducing x^k to x every time its degree becomes greater than one during any computational step. For example, the monomial $x_1^2x_2^3x_3$ is equal to $x_1x_2x_3$ in the Boolean domain.

By ordering each variable of the model according to its reverse topological level in the circuit, the generated polynomials satisfy Definition 3 by construction. Every polynomial is of the form $p_i := x_i + \mathrm{tail}(p_i)$, where x_i is the gate's output variable and $\mathrm{tail}(p_i)$ are terms consisting of the gate's input variables, describing the function implemented by the gate. According to this polynomial form, all leading monomials of the model are relatively prime.

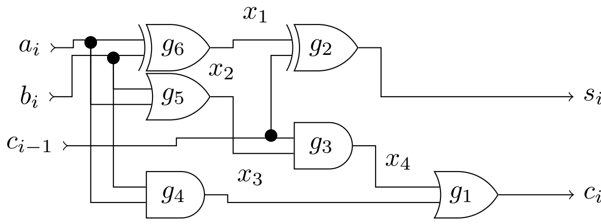

Fig. 1. A simple full adder.

Example 1: Consider the full adder circuit implementing the function $s_i + 2c_i = a_i + b_i + c_{i-1}$ shown in Fig. 1. Its algebraic model is

$g_1 := -c_i - x_4x_3 + x_4 + x_3$ $\quad g_2 := -s_i - 2x_1c_{i-1} + x_1 + c_{i-1}$
$g_3 := -x_4 + x_2c_{i-1}$ $\quad g_4 := -x_3 + a_ib_i$
$g_5 := -x_2 - a_ib_i + a_i + b_i$ $\quad g_6 := -x_1 - 2a_ib_i + a_i + b_i$

The specification polynomial[1] is $p_\mathrm{r} := -2c_i - s_i + c_{i-1} + b_i + a_i$. Ordering the polynomial variables in the reverse topological order of the circuit yields $c_i > s_i > x_4 > x_3 > x_2 > x_1 > c_{i-1} > b_i > a_i$. Following this order, the leading monomials of all polynomials will be relatively prime. E.g., the leading monomial of g_1 is c_i, and it is relative prime to all other leading monomials. According to Definition 3, the extracted algebraic model is therefore a Gröbner basis.

Modeling the circuit directly as Gröbner basis polynomials avoids Buchberger's algorithm and makes it computationally feasible to apply the membership testing.

C. Ideal Membership Testing

Given a specification (or relationship) polynomial p_r and a circuit model in form of a Gröbner basis G, p_r is divided in every iteration by some polynomial $g \in G$ (see Definition 1). The polynomial division can be seen as substituting the variables in p_r with the corresponding tail terms of the respective polynomials in G. For example, given $p_\mathrm{r} := x_4x_3 + x_1$ and a polynomial $g := -x_4 + x_2x_1$, then $\mathrm{r} = p_\mathrm{r} - \frac{x_4x_3}{-x_4}g = x_3x_2x_1 + x_1$, where the polynomial division substitutes x_4 in p_r with x_2x_1. The division (substitution) iterations are executed according to a certain order, the *substitution order*. As in [1], [9], the substitution ordering follows the reverse topological order of the circuit variables.

Following Example 1, the extracted algebraic model is a Gröbner basis, therefore the ideal membership testing of p_r can be applied. The substitution order will follow the reverse topological order of the circuit:

$$p_\mathrm{r} \xrightarrow{g_1} -s_i + 2x_4x_3 - 2x_4 - 2x_3 + c_{i-1} + b_i + a_i$$
$$\xrightarrow{g_2} 2x_4x_3 - 2x_4 - 2x_3 + 2x_1c_{i-1} - x_1 + b_i + a_i$$
$$\xrightarrow{g_3} 2x_3x_2c_{i-1} - 2x_3 - 2x_2c_{i-1} + 2x_1c_{i-1} - x_1 + b_i + a_i$$
$$\xrightarrow{g_4} 2x_2c_{i-1}b_ia_i - 2x_2c_{i-1} + 2x_1c_{i-1} - x_1 - 2b_ia_i + b_i + a_i$$
$$\xrightarrow{g_5} 2x_1c_{i-1} - x_1 + 4c_{i-1}b_ia_i - 2c_{i-1}a_i - 2c_{i-1}b_i - 2a_ib_i + b_i + a_i \xrightarrow{g_6} 0$$

Since the final division result is 0, p_r has been proven.

III. Algebraic Combinational Equivalence Checking

This section introduces the proposed algebraic combinational equivalence checking approach. Given two circuits C_1 and C_2 that represent the functions $f_1(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$ and $f_2(x_1, \ldots, x_n) = (z_1, \ldots, z_m)$, respectively, our aim is to show the equivalence of C_1 and C_2, i.e., $(y_1, \ldots, y_m) = (z_1, \ldots, z_m)$ for all $x_1, \ldots, x_n$. We propose to solve this problem using symbolic computation. Since the specification of C_1 and C_2 may be unknown or since it may not be expressible in a canonical and an abstract form over $\mathbb{Z}_{2^n}$, we cannot use previous work [1]–[3] that performs ideal membership testing with respect to a given specification.

Instead we propose to represent C_1 and C_2 as polynomial sets G_1 and G_2 and combine them into a single model $G = G_1 \cup G_2$. We then formulate the problem as testing the membership of relations between variables in C_1 and C_2 wrt. G. An obvious choice for such a relation is the equivalence of output signals $y_i = z_i$ which can be expressed in a polynomial as $y_i - z_i = 0$. However, reducing such a polynomial wrt. G causes a tremendous overhead since the substitution of all the internal variables in G_1 and G_2 will blow up the sizes of the polynomials in G.

To overcome this problem we suggest to find internal equivalences, i.e., polynomials that express equivalence of two internal signals in G_1 and G_2. Reducing these polynomials wrt. G causes a smaller overhead and simplifies G. This technique is similar to SAT sweeping in combinational equivalence checking [4] and we call it *arithmetic sweeping* in the following. Arithmetic sweeping works as follows: for each internal variable v_1 in G_1 we search for an equivalent variable v_2 in G_2, i.e., v_1 and v_2 represent the same function wrt. to the primary inputs. We call such a pair (v_1, v_2) *bit equivalence* and are able to substitute v_2 by v_1 in all polynomials. For some internal variables we will not be able to prove equivalence to another variable. These variables are eliminated by substitution with proved bit equivalent variables of their transitive fan-in.

However, performing arithmetic sweeping on the overall combined model G is not scalable. First, the number of candidates for bit equivalences is too large, and second, checking a pair of variables for equivalence that have a large transitive fan-in may be too difficult. To circumvent this problem, we first apply *reverse engineering* for two main goals i) extracting and abstracting arithmetic word-level components to canonical polynomials; ii) partitioning the circuits G_1 and G_2 into smaller parts. The algorithm works as follows: First, we try to find an instance of an arithmetic word-level component both in G_1 and G_2 and abstract them to canonical polynomials. If this is successful, we obtain an input boundary and an output boundary for the component in G_1 and G_2. The pairs of input boundaries and output boundaries are candidates for *word equivalences*. Having them, we perform arithmetic sweeping only in the transitive fan-in of the input boundaries. If this ultimately proves that the input boundaries are equivalent and we have proven that *abstracted polynomials* of the two arithmetic components found by reverse engineering are equivalent, we can merge the transitive fan-in of the output boundaries from G, making the model significantly smaller.

Details on the algorithms of our ACEC are explained in the remaining sections. In Section IV we show how to find arithmetic word-level components using reverse engineering. This also partitions the circuits into smaller parts based on the word-level components and the respective transitive fan-in of it. Both results are the input for the arithmetic sweeping. Section V explains arithmetic sweeping to find (and prove) internal equivalences in the transitive fan-ins of the detected

[1]Please note that later in the paper we use polynomials which relate different bit or word variables, so we call them relationship polynomials.

components' input boundaries. Finally in Section VI, we offer efficient polynomial representation based on functional decomposition and a new general reduction rule to speed up the different division steps.

IV. Reverse Engineering of Data-path Units

Key in ACEC is to find arithmetic components using reverse engineering in order to reduce the model size in which arithmetic sweeping is performed. Reverse engineering needs to find equivalent components and abstract them to canonical polynomials over integer field. The propagation of carry bits between internal nets of data-path units is one of the main properties that helps to locate such units. In the proposed reverse engineering algorithm we exploit this property to extract data-path units from the combined model G. According to our observation, these carry bits are modeled as shared monomials between polynomials of G. Continuing with Example 1, the simple full adder can be modeled by two polynomials $g_1 : -s_i+4c_{i-1}b_ia_i-2c_{i-1}b_i-2c_{i-1}a_i-2b_ia_i+c_{i-1}+b_i+a_i$ and $g_2 : -c_i - 2c_{i-1}b_ia_i + c_{i-1}b_i + c_{i-1}a_i + b_ia_i$. The shared monomials $c_{i-1}b_ia_i$, $c_{i-1}b_i$, $c_{i-1}a_i$, and b_ia_i model the internal carry propagations of the full adder. The terms of these shared monomials have another property. Their coefficients have different signs, and they are multiples of each other. We call terms with these properties *carry terms*.

To reveal carry terms, a rewriting scheme based on the rewriting principles of [3] is proposed in Section IV-A. Note that in the full adder example, carry terms are not visible in the original model of the full adder. They can be only revealed when the model is rewritten to two polynomials.

After rewriting the model, the reverse engineering algorithm builds an *adder network* for every group of polynomials that share carry terms. For each adder network which models a data-path unit *one* canonical polynomial using *Gaussian elimination* algorithm is derived, see Section IV-B.

A. Model Rewriting

The model rewriting schemes of [3] have shown an ability to reveal vanishing monomials (monomials that always evaluate to zero) as well as common monomials between the polynomials model of a multiplier circuit. This revealing ability empowers our reverse engineering algorithm to build adder networks for different architectures of large scale multipliers and adders. The proposed rewriting schemes combine the knowledge of the circuit gates with the algebraic model. The first scheme *XOR rewriting* rewrites the model using the S-polynomial method such that the model depends only on inputs and output variables of XOR gates whereas all other variables are substituted. The second *common rewriting* scheme rewrites the model obtained from XOR rewriting such that the model depends only on variables that are used in more than one polynomial.

Applying these schemes on a control logic circuitry causes a blow-up in the number of model terms since control logic usually does not contain XOR gates which yields to substitutions for large the number of the control logic variables. To take advantage of these schemes for circuits which contain data-path and control logic, we distinguish the control logic part of a circuit by its multiplexers (MUXes) and disallow XOR rewriting and common rewriting from substituting input and output variables of MUXes. This guarantees that both schemes will be applied only on the data-path logic. The polynomials of the rewritten model describe functions of XORs, MUXes, and the cone of gates which are bounded by inputs and outputs of XORs and MUXes.

B. Abstracting Data-path Units to Canonical Polynomial

After rewriting G the algorithm builds different adder networks from polynomials that share carry terms. It then groups polynomials of G. A new polynomial joins a group, if it shares a carry term with other polynomials in the group. For example, the polynomials g_0 and g_1 are in the same group, if one of them has the term $-x_0x_1$ and the second has the term $2x_0x_1$. Groups of each extracted adder network are handled as independent algebraic ideal. It is abstracted to one canonical polynomial using *Gaussian elimination*.

Example 2: To illustrate the proposed approach, consider the model of a 3-bit ripple carry adder implementing the function $\sum_{i=0}^{2} 2^i s_i = \sum_{i=0}^{2} 2^i(a_i + b_i)$.

$s_3 = c_2 \implies g_1 := -s_3 + c_2$
$c_2 = (a_2 \wedge b_2) \vee (a_2 \wedge c_1) \vee (b_2 \wedge c_1) \implies$
$g_2 := -c_2 \boxed{-2c_1b_2a_2 + c_1b_2 + c_1a_2 + b_2a_2}$
$s_2 = a_2 \oplus b_2 \oplus c_1 \implies$
$g_3 := -s_2 \boxed{+4c_1b_2a_2 - 2c_1b_2 - 2c_1a_2 - 2b_2a_2} + c_1 + b_2 + a_2$
$c_1 = (a_1 \wedge b_1) \vee (a_1 \wedge c_0) \vee (b_1 \wedge c_0) \implies$
$g_4 := -c_1 \boxed{-2c_0b_1a_1 + c_0b_1 + c_0a_1 + b_1a_1}$
$s_1 = a_1 \oplus b_1 \oplus c_0 \implies$
$g_5 := -s_1 \boxed{+4c_0b_1a_1 - 2c_0b_1 - 2c_0a_1 - 2b_1a_1} + c_0 + b_1 + a_1$
$c_0 = a_0 \wedge b_0 \implies g_6 := -c_0 \boxed{+b_0a_0}$
$s_0 = a_0 \oplus b_0 \implies g_7 := -s_0 \boxed{-2b_0a_0} + b_0 + a_0$

Rewriting the model yields that polynomials g_2, g_3 have common non-linear monomials (colored green/dashed box in the example). The similar structural property can be seen for equally colored terms of the polynomials g_4, g_5 and polynomials g_6, g_7, respectively. To cancel the carry terms between g_3 and g_2, Gaussian elimination is applied. It multiples g_2 by 2 and adds it to g_3. The result is the polynomial $h_1 := -2c_2 - s_2 \ + c_1 + b_2 + a_2$ which represents a full adder function. Applying the same step on other related polynomials yields another two full adders $h_2 := -2c_1 - s_1 \ + c_0 + b_1 + a_1$ and $h_3 := -2c_0 - s_0 \ + b_0 + a_0$. Applying Gaussian elimination again on the three full adder polynomials to cancel shared terms and achieve a reduced Gröbner basis, will multiply h_1 by 2 and adds to h_2. The result will be $h_4 := -4c_2 - 2s_2 - s_1 + 2b_2 + 2a_2 + c_0 + b_1 + a_1$. Finally, the reduced Gröbner basis polynomial $h_5 := -8c_2 - 4s_2 - 2s_1 - s_0 + 4b_1 + 4a_1 + 2b_1 + 2a_1 + b_0 + a_0$ is derived by multiplying h_4 by 2 and adding it to h_3 for canceling the shared monomial c_0.

Lemma 3: Let $G_r = g_1, \cdots, g_t$ denote the generated Gröbner basis by Gaussian elimination wrt. a unique monomial order $>$. As G_r contains the one and only polynomial g_1, then g_1 is the unique canonical representation of the function f implemented by the adder network ideal.

Proof: Based on Lemma 2, for every ideal there is a unique reduced Gröbner basis. Since the adder network ideal G_r, which has been generated by Gaussian elimination, has only one polynomial g_1, no term in g_1 is divisible by the leading term of any other polynomial in G_r. Therefore Definition 4 of reduced Gröbner basis holds for G_r and g_1 is a canonical abstracted representation of the function f implemented by the adder network ideal.

Please also note that to avoid the blow-up in the number of terms during Gaussian elimination of large scale multipliers,

as illustrated in [1], [9], the elimination order must follow the reverse topological order of the circuit variables.

In addition to abstracting data-path units, the reverse engineering algorithm determines their inputs and outputs boundaries. This works as follows: The algorithm extracts this information from the original polynomials (ideal) of the adder network. It uses a property of a Gröbner bases model that a variable of a leading monomial of a polynomial is the output variable of this polynomial. Based on this property, for the ideal of an adder network, output variables of polynomials that are not used as inputs for other polynomials in the ideal are identified as the output variables of this adder network. Finally, an output word for each abstracted polynomial can be derived.

V. Arithmetic Sweeping

Arithmetic sweeping aims to find internal equivalences which avoids prohibitive run time during the polynomial division. Of course when having identified candidates for internal equivalence, it is still necessary to prove their equivalence (which is also done using the same division algorithm for the relationship polynomials of the candidates). Hence, to gain an overall benefit we need i) promising candidates and ii) moderate runtimes for the equivalence proofs. Our proposed arithmetic sweeping reaches both goals as follows.

For i), the reverse engineering step provides arithmetic components. From this we generate promising candidates based on the I/O boundaries of these components. The algorithm uses the I/O boundaries to partition the variables of the combined model G into groups. Simulation deduces *word equivalence* (wE; for details see below) candidates between outputs of the arithmetic components. For every nominated wE the partitioning of model variables is performed by classifying two groups of variables. One for the transitive fan-ins variables of the input boundaries of wE and the other are internal variables of the two related arithmetic components. Deducing only internal *bit equivalences* (bE; see below) between variables in the same group increases the potential of equivalence.

For ii), the equivalence proofs become feasible for several reasons. Arithmetic sweeping generates two types of relationships which are *bit equivalence* (bE) pair and *word equivalence* (wR) pair. bE describes the equivalence of a pair of variables (v_i,v_j) and is formulated by the polynomial $g := -v_i + v_j$. The word equivalence (wE) polynomial is formulated as $g := B - \hat{B}$ for the word pair candidate (B,$\hat{B}$), where $B = 2^{n-1}b_{n-1} + \cdots + b_0$ and $\hat{B} = 2^{n-1}\hat{b}_{n-1} + \cdots + \hat{b}_0$. For each arithmetic component we have determined an *abstract* canonical polynomial in the reverse engineering step. The major advantage over SAT sweeping is that the proof for the internal equivalences is performed by dividing wE polynomials wrt. the abstracted polynomials. For doing this, a new word model G_W is created and the abstracted polynomials are added to it as follows: For every abstracted polynomial, an integer word variable B is created and a polynomial $-B + f(a_1, \cdots, a_m)$ is added to the word model. The polynomial $-B + 2^{n-1}b_{n-1} + \cdots + b_0$ is used to interpret the equivalence between two output words B and $\hat{B}$, as shown in Lemma 4 of Section V-B. To summarize, dividing wE wrt. abstracted polynomials has a major influence on the performance of the technique—it avoids the exhaustive cost of searching for equivalences between internal variables of the data-path units which usually have a large the number of non-equivalent variables in their transitive fan-ins.

A. Generating Relationship Polynomials

The choice of relationship candidates is always the main problem of different equivalence checking techniques. ACEC draws on the simulation approach of [10] and the extracted data-path polynomials to deduce bit and word relationships. Four steps are performed to generate relationship polynomials i) nominating wE polynomials, ii) classifying the model variables to groups, iii) generating bE polynomials, and finally iv) sorting wE and bE polynomials in a relationship list.

Based on a fixed size of global simulation over the primary inputs of G, word relationships between the output words of the data-path polynomials are deduced. Two words build a wE polynomial, if their integer values are equal under all simulated assignments.

The approach classifies the variables of G to groups according to wE polynomials. One wE polynomial categorizes two groups, the first consists of all transitive fan-in variables of the polynomial; and the second contains internal variables which are bounded by outputs and inputs variables of wE.

Example 3: To illustrate this idea, consider a model which has four extracted data-path units (DPU_1, DPU_2, DPU_3, and DPU_4), as shown in Fig. 2. The simulation nominates two wEs, one relates the output word of DPU_1 and DPU_2, the other one is between DPU_3 and DPU_4. The approach classifies the model variables into 5 groups i) a group for transitive fan-in variables of DPU_1 and DPU_2, ii) a group which contains internal variables of DPU_1 and DPU_2, iii) transitive fan-in variables of the wE between DPU_3 and DPU_4, iv) their internal variables, and v) the remaining variables of C_1 and C_2 which are not classified in groups.

Classified groups of G and global simulation are used to determine for every model variable v_i a set of variables ϕ_i. We have $v_j \in \phi_i$, if Boolean values of v_i and v_j are the same under each of input assignments; and therefore v_i and v_j belong to the same classified group. Finally, bE polynomials between v_i and other variables of ϕ_i are generated. We call them *bE polynomials* of the variable v_i.

After classifying model variables and generating wE and bE relationships, these nominated relationships are sorted topologically wrt. the circuit and their leading variables. The sorting procedure aims to test a wE polynomial after testing all bE polynomials of variables in its transitive fan-in group. First, the wE polynomials are sorted topologically. Next, the procedure iterates over the wE polynomials for inserting in the list for every wE i) bE polynomials of variables in the transitive fan-in group of this wE, ii) the wE polynomial itself, then iii) bE polynomials of variables in its internal group. Finally, the bE polynomials of remaining variables that are not included in groups that are related to wEs, are inserted in the end of the list.

B. Testing Membership of Internal Relations

During the testing of internal relationships, the approach calls the IMT algorithm to divide every polynomial p_r from the relationship list wrt. G or G_W, if p_r is a bE polynomial, the division is done wrt. G, otherwise is performed wrt. G_W. Based on the remainder result of dividing p_r, the approach

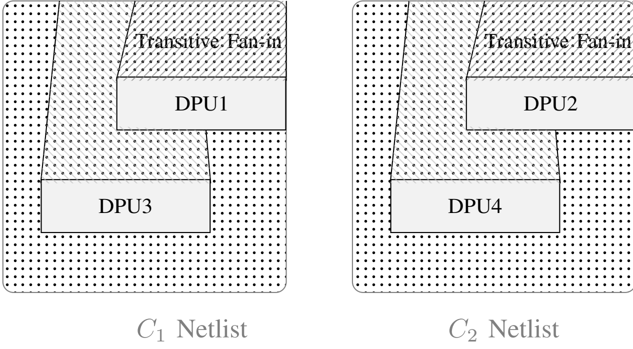

Fig. 2. Schematic of a combined model including word relationships

eliminates or merges variables of p_r from the models G and G_W.

The merging decision is taken, in the case that the remainder result of dividing p_r is equal to zero. The approach merges every two variables of p_r which are derived to be functionally equivalent to one variable. In case that p_r is a wE polynomial, equivalence is derived based on the following lemma.

Lemma 4: Given the equivalent of two integer words $A = 2^{n-1}a_{n-1} + \cdots + a_0$ and $B = 2^{n-1}b_{n-1} + \cdots + b_0$. If A and B have the same number system and the number system is not redundant, then the bit variables a_i and b_i which have same weights (coefficients) are equivalent.

Merging results are reducing the number of polynomials in G, these merged variables are considered new primary inputs, therefore polynomials of their transitive fan-in variables are removed from G. Continuing with the previous model example, after deriving the equivalences between corresponding output variables of DPU_1 and DPU_2, merging equivalent variables will produce a compact version of G. Polynomials that model DPU_1 and DPU_2 are removed, in addition to those which model their transitive fan-in variables. In order to avoid redundant divisions, the remaining bE polynomials which test the already merged variables will be removed from the relationship list.

A variable of the model that has no functional equivalences is eliminated by substituting it with the leading terms of its polynomials which are functions in proved bit equivalent variables. The elimination decision will be taken for variables v_i of p_r. If the remainder of dividing p_r is not equal to zero, and there are no more untested bE or wE polynomials in the list which are related to v_i. These eliminations facilitate the division process of next relationships. It increases the number of shared input variables of polynomials of G which simplifies the division process of p_r wrt. G. For example, dividing $p_r : -v_i + v_j$ wrt. a model that has polynomials $g_1 : -v_i + x_1x_2 + x_3$ and $g_2 : -v_j + x_1x_2 + x_3$ will be simplified to a subtraction operation. The remainder of the division will be $x_1x_2 + x_3 - x_1x_2 - x_3 = 0$.

VI. Efficient Polynomial Representation

The polynomial is the heart of the algebraic computation technique. An efficient representation of a polynomial has a major impact on the performance of any algebraic algorithm. To circumvent a blow-up in the number of polynomial terms for representing different Boolean functions, we propose i) a decomposition method which reduces the number of terms in polynomials significantly for some Boolean functions, and ii) a general reduction rule to cancel redundant terms of the polynomial. The decomposition method and the reduction rule offer semi-canonical representations which simplify the division of the IMT algorithm.

A. Different Decompositions

Inspired by decomposition types of decision diagrams [11], we enhance representations of polynomials by considering two decomposition types which are:

$$\begin{array}{ll} f = f_{|x=0} + x(f_{|x=1} - f_{|x=0}) & \text{positive Davio (pD)} \\ f = f_{|x=1} + (1-x)(f_{|x=0} - f_{|x=1}) & \text{negative Davio (nD),} \end{array}$$

where x denotes a Boolean variable, the functions are combined with addition, subtraction, and multiplication operations.

Our observation is that polynomials have been typically represented by pD decomposition. This obstructs a compact representation for some Boolean functions like a chain of OR gates. For example, consider a 4-input OR function $f(x_0, x_1, x_2, x_3)$, its polynomial representation that follows only pD will be $f = x_0 + x_1 + x_2 + x_3 - x_0x_1 - x_0x_2 - x_0x_3 - x_1x_2 - x_1x_3 - x_2x_3 + x_0x_1x_2 + x_0x_1x_3 + x_0x_2x_3 + x_1x_2x_3 - x_0x_1x_2x_3$. By decomposing f using nD for all of its variables, it will be $f = 1 - \bar{x}_0\bar{x}_1\bar{x}_2\bar{x}_3$, where $\bar{x}_i = 1 - x_i$. For the n-bit OR function, a polynomial which follows pD consists of $2^n - 1$ terms, while nD polynomial has only two terms. Representing a Boolean function with less the number of terms has a major influence on reducing the number of addition, subtraction, and multiplication operations, therefore it enhances significantly the performance of any symbolic computation algorithm. For applying these decompositions, we add to the model negation version $\bar{v}_i$ for every variable v_i in the model, in addition to the polynomials $g := -\bar{v}_i - v_i + 1$.

As known from the field of decision diagrams, the choice of the type of the decomposition and the order of the variables plays a key role for the size of the diagram. In this work, we fix the order of the variables to the reverse topological order and we propose an approach to determine the *Decomposition Type* (DT) of each variable. As the main goal of applying different decompositions is reducing the number of polynomials terms, the decision of DT is taken based on this factor and the structure of the circuit.

For this purpose, we modify the modeling way of the circuit that is explained in Section II-B as follows:

$$\begin{aligned} z = \neg a &\implies g := -z + \bar{a} \\ z = a \wedge b &\implies g := -z + ab \\ z = a \vee b &\implies g := -z + 1 - \bar{a}\bar{b} \\ z = a \oplus b &\implies g := -z + a + b - 2ab, \end{aligned}$$

such that the DT of input variables of inverters and OR gates is nD, for AND and XOR gates, it is pD. As shown in this modeling, one variable may have more than one DT in the model. During model rewriting, see Subsection IV-A, a polynomial g is rewritten by substituting one of its variables v_i, as result of this step, another variable v_j in g may have different decomposition types – the variable v_j and its negation $\bar{v}_j$ are within the same polynomial g. In this case, we unify the DT of v_j based on the one which achieves the higher reduction on number of terms in g. In case of n variables with different DTs within same polynomial, the possible combinations of DTs for these variables are 2^n. For example, consider a polynomial with two variables v_1 and v_2, the possible decomposition

combinations will be (v_1, v_2), $(v_1, \bar{v_2})$, $(\bar{v_1}, v_2)$, or $(\bar{v_1}, \bar{v_2})$. Trying all combinations to find the best representation leads to a prohibitive run time because of calling the decomposition algorithm 2^n times. To bypass this problem, our approach takes the decomposition decision of every variable independently from others. This restriction on choosing DTs accelerates significantly the run time of the proposed approach to find compact representations for polynomials. In this work, we have used an implemented decomposition algorithm which designed for decision diagrams [12]. For this, we have implemented a two directions parser. It parses a function from a polynomial to a K*BMD and vice versa.

B. General Reduction Rule

A key observation in [3] is the significance of applying a logic reduction rule to cancel redundant terms and avoid blow-up of these terms during the division algorithm. The rule exploits that $(a \oplus b) \cdot (a \wedge b) = 0$ for all a and b and therefore can be used to remove terms from polynomials. If, e.g, $f = a \oplus b$ and $h = a \wedge b$, any term containing both f and h can be removed. We propose to generalize this rule.

Let X be a set of variables and let f and h be two Boolean functions over the variables $X_1 \subseteq X$ and $X_2 \subseteq X$, respectively, with $X_1 \cap X_2 \neq \emptyset$. If there exists exactly one assignment to h such that it evaluates to true, it may be possible that g simplifies to a constant value when assigning the common variables according to that assignment. To illustrate the concept consider a multiplexer function $f(a, b, c) = ac - bc + b$ and $h(a, b) = ab$. Clearly $h = 1$, only if $a = b = 1$, and $f(1, 1, c) = 1$. Therefore, we conclude that $fh = h$ and we can simply polynomials accordingly. For a polynomial $g := -v_1 v_2 f h v_3 + v_1 v_2 h v_3$, by applying this rule on the monomial $v_1 v_2 f h v_3$, it simplifies to $v_1 v_2 h v_3$ and the polynomial g will be evaluated to $g = -v_1 v_2 h v_3 + v_1 v_2 h v_3 = 0$. This reduction rule is called *one assignment rule*.

An approach to apply the one assignment rule is as follows:

1) Searching for monomials in the algebraic model that have two variables of functions f and h which shared some of their inputs, such that the function h has one satisfiable assignment.
2) Reducing f after assigning values to shared inputs which evaluate h to one.
3) If f is equal to zero or one, then rewriting the monomial by substituting f with its value.

VII. Experimental Evaluation

ACEC is implemented in C++. We compared it to the equivalence checkers of ABC [13] tool and a commercial tool (OneSpin EC-360). The experiments were carried out on an Intel(R) Core(TM) i5-3320M CPU (2.6 GHz, 16 GByte) running Linux.

We applied ACEC to the problem of verifying a floating-point (FP) multiplier. It computes the operation $P = A \times B$ for two FP operands $A = (-1)^{s_a} \times 2^{e_a} \times f_a$ and $B = (-1)^{s_b} \times 2^{e_b} \times f_b$. s_a denotes the sign, e_a the exponent, and f_a the significand including the implicit bit of the operand A (similarly for B and P). The operation can be defined as $s_p = s_a \oplus s_b$ and $2^{e_p} \times f_p = RND(2^{e_a + e_b} \times f_a \times f_b)$. RND is the round and normalize function according to the IEEE standard for floating-point arithmetic (IEEE Std 754-2008).

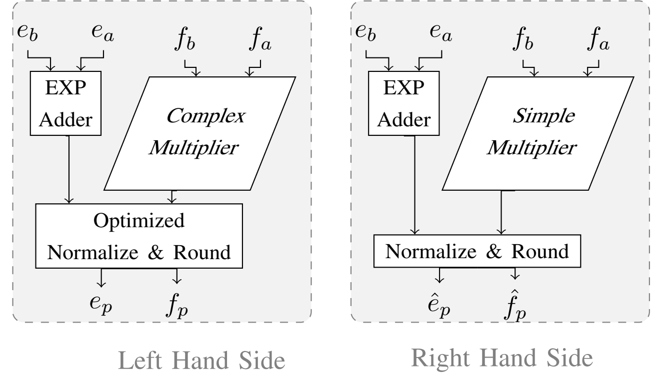

Fig. 3. Compared FP Multiplier Circuits

We have scaled and modified the structure of the FP multiplier unit of the open cores design module DOUBLE-FPU [14] for building dissimilar FP instances. As shown in Fig. 3, the compared circuits have different multiplier architectures and their control logic units are optimized distinctively. The multiplier units are generated using the online tool *Arithmetic Module Generator* [15]. These generated circuits[2] were synthesized from Verilog to gate level netlists using *Yosys* [16].

The multiplier architectures are categorized according to 1) the type of the partial products generator, 2) the partial products accumulator, and 3) the last stage adder. In our experiments, we use a partial products generator, namely *simple partial products* (SP). The types of partial products accumulators are *array* (AR), (4,2) *compressor tree* (CT), and *wallace tree* (WT). The last stage adder are *ripple carry adder* (RC), *carry look-ahead adder* (CL), and *brent-kung adder* (BK).

In Table I, we demonstrate the runtimes of checking the equivalences of divergent FP multipliers against the same circuit reference. The reference consists of a simple multiplier (SP-AR-RC) and unoptimized normalize round unit. While the compared circuits contain complex multipliers and round units which are optimized using the Yosys option *share*[3]. The first column of Table I shows the type of the multiplier architecture. The second and the third columns give the number of bits of an FP operand of the circuit in addition to the size of its significand and its exponent according to the IEEE standard. The next three columns provide the runtimes. The timeout (TO in the table) is set to 24 hours. The experimental results clearly demonstrate the advantage of ACEC in verifying circuits that include data-path and control logic. While other equivalence checking tools can verify the correctness up to 16 bits, we are able to verify the correctness of a single precision binary floating-point multiplier (32 bit).

Table II shows some statistics about the algorithms of ACEC for checking the equivalence of the FP multiplier instances that contain the multiplier architecture SP-WT-CH. For the reverse engineering algorithm, it shows the runtime of rewriting the combined model G; the runtime of extracting and abstracting data-path units; and the number of the extracted units. These results show that the reverse engineering algorithm extracts

[2]The benchmarks, binary of our tool, and log files are available at http://www.informatik.uni-bremen.de/agra/eng/asc.php

[3]It merges shareable resources into a single resource. A SAT solver is used to determine if two resources are shareable

TABLE I
RUNTIMES FOR CHECKING FP MULTIPLIERS EQUIVALENCES

Multiplier Architecture	FP operand # bits	Significand/Exponent # bits	Commercial (h:m:s)	ABC (h:m:s)	ACEC (h:m:s)
SP-CT-BK	16	13/3	00:08:50	TO	00:01:42
SP-WT-CH	16	13/3	00:09:08	TO	00:01:44
SP-CT-BK	24	21/3	TO	TO	00:17:49
SP-WT-CH	24	21/3	TO	TO	00:25:58
SP-CT-BK	32	25/7	TO	TO	02:24:01
SP-WT-CH	32	25/7	TO	TO	03:41:43

TABLE II
STATISTICS OF ACEC FOR EQUIVALENCE CHECKING OF FP MULTIPLIERS

# bits	ACEC Algorithms		
	Reverse Engineering		
	Model Rewriting (h:m:s)	Extract & Abstract (h:m:s)	# Data-path Units
16	00:00:46	00:00:23	21
24	00:11:56	00:10:04	23
32	00:32:50	02:10:30	23
	Arithmetic Sweeping		
	# Variables of G	# Proved Equivalences	Runtime (h:m:s)
16	1888	401	00:00:27
24	4440	666	00:03:36
32	5889	854	00:58:04
	Efficient Polynomial Representation		
	Decomposition # Reduced Terms	Decomposition # Eff./Total Calls	Logic Reduction # Canceled Terms
16	2400	514/3477	2916
24	9732	1013/8684	17153
32	16317	1345/12477	36390

more candidates for data-path units than the expected number. For two combined FP multipliers, six data-path units should be extracted, two significand multipliers, two exponent adders, and two incrementers in the rounding stages. Also, the results show that most of the run-time of ACEC is spent in reverse engineering (on average about 65%).

For arithmetic sweeping, Table II gives total the number of variables of the combined model; the number of proved equivalences between variables of the two compared circuits; and the time spent by the sweeping algorithm. The results demonstrate that variables of G which have functional similarities between each other account for less than 45% of the total the number of variables. Further, the table shows the number of saved terms by the decomposition of polynomials; the number of effective (Eff.) calls for the decomposition algorithm wrt. the total calls for the algorithm (effective calls are those which save terms of polynomials), and the number of canceled terms by the reduction rule.

VIII. RELATED WORK

One noteworthy challenge is developing a fully automated technique which proves that a floating-point design is in consistence with the IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2008). Theorem provers have been applied extensively to verify the properties of floating-point designs. Although a lot of automation has been added and floating-point libraries have been created to avoid repetition of proofs, theorem proving methodology still requires an enormous amount of manual effort, expert knowledge, and high understanding of the design [17]. The paper by Jacobi [18] is the most automated work up to today, however, it skips the hardest part to verify, the multiplier.

As mentioned already in the paper all the existing works (e.g. [1]–[3], [9]) using Gröbner bases for circuit verification only target pure arithmetic components w/o control logic.

The recently proposed reverse engineering algorithms [19], [20] for the extraction of arithmetic word level components from a gate-level netlists are not applicable to designs with a non-arithmetic combinational logic attached to the output.

IX. CONCLUSION

In this paper we have presented a new algebraic equivalence checking technique for checking the equivalence of circuits that combine data-path and control logic. The technique utilizes a new reverse engineering algorithm to extract and abstract arithmetic components from the combined model of the Gröbner bases representation of the compared circuits. Based on input and output boundaries of the abstracted components the proposed arithmetic sweeping deduces less and promising candidates for bit and word equivalences between the compared circuits. The technique circumvents the blow-up in the number of terms of polynomials during the utilized algorithms by offering different types of decompositions for polynomials and using an efficient reduction rule. Experimental results demonstrated the efficiency of our technique for the equivalence checking of large floating-point multipliers which cannot be verified with existing Boolean combinational equivalence checking techniques.

For future work we want to investigate canonization for control logic as well as managing the membership testing for non-equivalent circuits.

REFERENCES

[1] F. Farahmandi and B. Alizadeh, "Gröbner basis based formal verification of large arithmetic circuits using Gaussian elimination and cone-based polynomial extraction," *MICPRO*, vol. 39, no. 2, pp. 83–96, 2015.

[2] T. Pruss, P. Kalla, and F. Enescu, "Efficient symbolic computation for word-level abstraction from combinational circuits for verification over finite fields," *TCAD*, vol. 35, no. 7, pp. 1206–1218, 2016.

[3] 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*, 2016, pp. 1048–1053.

[4] A. Kühlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *TCAD*, vol. 21, no. 12, pp. 1377–1394, 2002.

[5] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *ICCAD*, 2006, pp. 836–843.

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

[7] Y. Sato, S. Inoue, A. Suzuki, K. Nabeshima, and K. Sakai, "Boolean Gröbner bases," *Journal of symbolic computation*, vol. 46, no. 5, pp. 622–632, 2011.

[8] A. Nagai and S. Inoue, "An implementation method of boolean Gröbner bases and comprehensive boolean Gröbner bases on general computer algebra systems," in *International Congress on Mathematical Software*, 2014, pp. 531–536.

[9] M. Ciesielski, C. Yu, D. Liu, and W. Brown, "Verification of gate-level arithmetic circuits by function extraction," in *DAC*, 2015, pp. 52:1–52:6.

[10] E. Goldberg, M. Prasad, and R. Brayton, "Using SAT for combinational equivalence checking," in *DATE*, 2001, pp. 114–121.

[11] R. Drechsler and D. Sieling, "Binary decision diagrams in theory and practice," *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 112–136, 2001.

[12] R. Drechsler, M. Herbstritt, and B. Becker, "Grouping heuristics for word-level decision diagrams," in *ISCAS*, vol. 1. IEEE, 1999, pp. 411–414.

[13] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *CAV*, 2010, pp. 24–40.

[14] D. Lundgren, "Double precision floating point core verilog," available at http://opencores.org/project,double_fpu, 2016.

[15] "Arithmetic module generator based on ACG," available at http://www.aoki.ecei.tohoku.ac.jp/arith/, 2016.

[16] C. Wolf, "Yosys open synthesis suite," available at http://www.clifford.at/yosys/, 2016.

[17] A. Slobodová, *Challenges for formal verification in industrial setting*. Springer Berlin Heidelberg, 2007.

[18] C. Jacobi, k. Weber, V. Paruthi, and J.Baumgartner, "Automatic formal verification of fused-multiply-add FPUs," in *DATE*, 2005, pp. 1298–1303.

[19] M. Soeken, B. Sterin, R. Drechsler, and R. Brayton, "Simulation graphs for reverse engineering," in *FMCAD*, 2015, pp. 152–159.

[20] C. Yu and M. Ciesielski, "Automatic word-level abstraction of datapath," in *ISCAS*, 2016, pp. 1–6.