Combining Symbolic Computer Algebra and Boolean Satisfiability for Automatic Debugging and Fixing of Complex Multipliers

Alireza Mahzoon[1] Daniel Große[1,2] Rolf Drechsler[1,2]
[1]Faculty of Mathematics and Computer Science, University of Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{mahzoon,grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—If verification of a digital circuit fails, then debugging and fixing become the major subsequent tasks. Arithmetic units are among the most challenging circuits for debugging because of a wide variety of architectures and high design complexity. A prominent example are multipliers. Since existing automatic methods fail for these circuits, both tasks are performed manually which is typically very time-consuming.**

In this paper, we propose a complete debugging flow based on the combination of Symbolic Computer Algebra (SCA) and Boolean Satisfiability (SAT). Complete means that our method targets the complete loop until the arithmetic circuit is guaranteed to fulfill its specification. For this, our approach consists of the three phases verification, localization, and fixing. In the experimental evaluation, we demonstrate the applicability of our approach for the most complex multiplier architectures.

I. Introduction

Nowadays, arithmetic circuits play a crucial role in many computation intensive applications (e.g. signal processing and cryptography) as well as in upcoming AI architectures (e.g. for machine learning and deep learning). At the heart of these arithmetic circuits integer multipliers and adders are the dominant building blocks. Due to the growing importance of power, speed, and area in digital circuits, designers have proposed a large variety of different integer multiplier and adder architectures to meet the pressing requirements. These architectures are usually extensively parallel and hence very complex. This makes them prone to design errors.

Since the famous Pentium bug back in 1994, a lot of effort has been put in the development of suitable verification methods. Ensuring the correctness in a mathematical sense became possible by *formal verification*. However, it is very well known that non-trivial arithmetic, and in particular integer multiplication at the gate level, is still one of the biggest challenges for formal methods. Looking from the methods perspective on formal verification, essentially five directions can be distinguished: (a) *Decision Diagrams* (DDs) (such as BDDs or *BMDs), (b) *Boolean Satisfiability* (SAT) and *Satisfiability Modulo Theories* (SMT), (c) reverse engineering techniques, (d) term rewriting, and (e) *Symbolic Computer Algebra* (SCA).

Considering an advanced gate level multiplier circuit as a representative input for each method, we can observe: DDs suffer from an exponential blow-up, SAT/SMT stuck for input datawidth greater than 15 bits, reverse engineering approaches using *Arithmetic Bit-Level* (ABL) representations [1], [2] can only handle simple multiplier architectures, and term rewriting techniques [3], [4] suffer from incompleteness and may fail to prove the correctness of the circuit because of insufficient lemmas. In contrast, SCA uses a polynomial representation for the problem. To be more precise, the circuit specification is represented as a single polynomial p_{spec} and the circuit is captured as a set of polynomials G. Then, the verification is done by testing the membership of the specification polynomial in an ideal with generators in G. This membership test corresponds to a series of divisions of p_{spec} by the circuit polynomials G (also known as Gröbner basis reduction). In the recent years, there was a renewed interest in SCA because these algebraic techniques have been applied successfully on large Galois Field arithmetic circuits [5], [6] as well as large (but architectural simple) integer multipliers [7], [8]. Recently, SCA has been shown to be very successful also for complex integer multipliers [9], [10], i.e. highly parallel architectures with up to 256 output bits. Basically, the authors of [9] have presented a logic reduction rewriting scheme consisting of XOR-rewriting and common-rewriting. This scheme allows for cancellation of so-called *vanishing monomials* (monomials which finally reduce to zero) in an efficient way *before* their blow-up during division. However, when verification fails, debugging and fixing the gate level arithmetic circuit are the next two major tasks. Hence, the designer has to (1) find the exact location of the bug and (2) determine a concrete fix.

In this paper, we propose an **approach for automatic debugging and fixing of complex gate-level arithmetic circuits combining SCA and SAT**. At first, we define the fault model. Subsequently, we show the limitations of a pure SCA-based method for debugging and fixing of complex arithmetic circuits. Then, we introduce our approach which employs an combination of SCA and SAT to successfully debug and fix arithmetic circuits. Our approach consists of three phases: verification, localization, and fixing. In each phase we employ SAT and SCA for individual subtasks as both have pros and cons. We explain the underlying decisions wrt. the chosen method which are also confirmed in the experiments. Finally, we show in the experimental evaluation on a very large set of multiplier circuits – ranging from simple to the most complex architectures – that automatic debugging and fixing is possible in practical time.

II. Related Work

Automated debugging using SAT has been initially presented in [11]. The approach introduces abnormal predicates into the netlist and allows to compute a list of suspicious locations (gates). Several improvements based on iterative analysis, abstraction, incremental SAT-solving, and better accuracy have been developed, see e.g. [12], [13], [14], [15]. However, these approaches can only be used for the localization step in the considered setting, since SAT/SMT is not able to perform the proof of correctness for complex arithmetic circuits, and therefore the verification of a fix fails.

Only a few debugging approaches using SCA have been proposed. In [16] the circuit is cut into levels and then based on the polynomial modeling backward and forward rewriting is performed simultaneously to identify differences. In contrast, the authors of [17] use the remainder of Gröbner basis reduction to find the location of the bug. However, both

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SELFIE under grant no. 01IW16001, by the University of Bremen's graduate school SyDe funded by the German Excellence Initiative, and by the German Academic Exchange Service (DAAD).

methods are limited to simple arithmetic circuits and they fail if vanishing monomials appear in the final remainder, or the bug is close to the *Primary Outputs* (POs). The proposed method in [18] is an extension of [17] and tries to debug faults close to POs. It uses partitions of the primary inputs' space of the design and performs incremental equivalence checking using Gröbner basis reduction. However, it still suffers from the vanishing monomial problem; we provide more details on this problem in Section IV-B. In summary, the existing approaches are not suitable for automated debugging and fixing of complex arithmetic circuits. Hence, in this paper we propose a combination of SCA and SAT to overcome the described limitations.

III. Preliminaries

In this section, first the concepts of SCA are described. Then, the process of verifying arithmetic circuits using SCA is reviewed.

A. Notations and Definitions

Definition 1: A *Monomial* is the product of variables in the following form

$$x^{\alpha} = x_1^{\alpha_1} x_2^{\alpha_2} \dots x_n^{\alpha_n} \tag{1}$$

Definition 2: A *Polynomial* is the finite combination of monomials with coefficients in k

$$f = \sum_{\alpha} a_{\alpha} x^{\alpha}, \quad a_{\alpha} \in k \tag{2}$$

The set of all polynomials with coefficients in k is denoted by $k[x_1, \dots, x_n]$.

The monomials of a polynomial are ordered based on the ordering of the variables and their powers. We use $A > B$ to show that A is in a higher order than B. For example, if there is $P = x^2y^3z^2 + x^3yz^2 + y^5z^2$, and the ordering of variables is $x > y > z$, then the ordering of monomials would be $x^3yz^2 > x^2y^3z^2 > y^5z^2$. The first monomial after ordering is called *Leading Monomial*, and denoted by $LM(P)$.

Definition 3: A subset $I \subset k[x_1, \dots, x_n]$ is an *ideal* if:

- $0 \in I$.
- if $f, g \in I$, then $f + g \in I$.
- if $f \in I$ and $h \in k[x_1, \dots, x_n]$, then $hf \in I$.

Consequently, if $f_1, \dots, f_s$ are polynomials in $k[x_1, \dots, x_n]$, then an ideal is generated by them in the following form

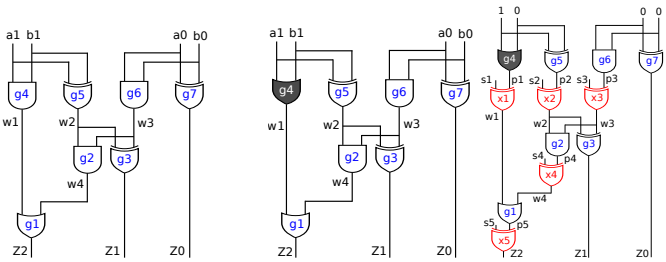
$$I =< f_1, \dots, f_s >= \{\sum_{i=1}^{s} h_i f_i : h_1, \dots, h_s \in k[x_1, \dots, x_n]\} \tag{3}$$

where $f_1, \dots, f_s$ are called *Generators* of the ideal.

Ideal Membership is one of the well-known problems in SCA. In this problem, a polynomial $f \in k[x_1, \dots, x_n]$ and an ideal $I =< f_1, \dots, f_s >$ are given, and the task is to determine if $f \in I$. In order to prove that $f \in I$, the remainder of dividing f by $f_1, \dots, f_s$ independent of division order should be always equal to zero.

Theorem 1: Let $g_1, \dots, g_s$ be the generators of an ideal I. The remainder of dividing f by the generators is always unique, if $LM(g_1), \dots, LM(g_s)$ are relatively prime. In other words, there should be no common variable in the leading monomial of the generators. The set of generators with this property is called *Gröbner basis* [19].

Note that Theorem 1 is already a special case of defining a Gröbner basis which facilitates the process of identifying a Gröbner basis in the context of circuits (for general case we refer to. [20]). Now, assume that $G = \{g_1, \dots, g_s\}$ is a *Gröbner basis*. Thus, the remainder r of dividing f by $g_1, \dots, g_s$ is uniquely determined, and the condition $r = 0$ is equivalent to membership in the ideal $I =< g_1, \dots, g_s >$.



(a) bug-free (b) buggy at gate g_4 (c) inserted XORs

Fig. 1. 2-bit adder circuit

The division is denoted by $f \xrightarrow{G} r$. For example, if $f = xz$, $g_1 = x + y$, and $g_2 = yz$ then $xz \xrightarrow{g_1} -yz \xrightarrow{g_2} 0$. To perform the division of xz by g_1, first g_1 is multiplied by z to create the same leading monomial xz as f, so $g_1 \times z = xz + yz$. Then, the subtraction is performed, i.e. we compute $f - (g_1 \times z) = xz - (xz + yz) = -yz$, which is the result of the first division. Finally, this result is divided by g_2 to obtain the remainder 0.

B. Verification using SCA

In verification of gate level arithmetic circuits, the specification polynomial of the circuit and a gate level netlist are provided as inputs, and the goal is to formally prove that they are equivalent. The specification polynomial of an arithmetic circuit determines the function of circuit based on its inputs and outputs. For example, for the 2-bit adder in Fig. 1a the specification polynomial is $F := 4Z_2 + 2Z_1 + Z_0 - (2a_1 + a_0 + 2b_1 + b_0)$ where $Z = 4Z_2 + 2Z_1 + Z_0$ shows 3-bit output, and $2a_1 + a_0 + 2b_1 + b_0$ indicates addition of 2-bit inputs.

Furthermore, each logical gate can be presented by a polynomial with coefficients in $\mathbb{Z}$ determining the relation between its inputs and output. The polynomials of basic Boolean gates are

$$\begin{aligned} z = \neg a &\Longrightarrow g := z - 1 + a & z = a \vee b &\Longrightarrow g := z - a - b + a \times b \\ z = a \wedge b &\Longrightarrow g := z - a \times b & z = a \oplus b &\Longrightarrow g := z - a - b + 2a \times b \end{aligned} \tag{4}$$

For example, if the variables in the 2-bit adder of Fig. 1a are ordered based on the reverse topological order of the circuit (from outputs toward inputs), the gates polynomials are

$$\begin{aligned} g_1 &:= Z_2 - w_1 - w_4 + w_1 w_4 & g_4 &:= w_1 - a_1 b_1 \\ g_2 &:= w_4 - w_2 w_3 & g_5 &:= w_2 - a_1 - b_1 + 2a_1 b_1 \\ g_3 &:= Z_1 - w_2 - w_3 + 2w_2 w_3 & g_6 &:= w_3 - a_0 b_0 \\ & & g_7 &:= Z_0 - a_0 - b_0 + 2a_0 b_0 \end{aligned} \tag{5}$$

The leading monomial of all gates polynomials are relatively prime. Therefore, based on Theorem 1 the set of gates polynomials is a *Gröbner basis*. Consequently, the problem of proving equivalency of the specification polynomial and the gate level netlist can be translated to a membership testing problem. In other words, if F (specification of a circuit) is a member of $< g_1, \dots, g_s >$ where $g_1, \dots, g_s$ are the gates polynomials of the the circuit, then the circuit is bug-free. The steps of dividing F by $g_1, \dots, g_7$ in the 2-bit adder of Fig. 1a is shown in (6). Due to the fact that the final remainder is equal to zero, the adder is correct.

$$\begin{aligned} F &\xrightarrow{g_1} F_1 := 4w_1 + 4w_4 \boxed{-4w_1w_4} + 2z_1 + z_0 - (2a_1 + a_0 + 2b_1 + b_0) \\ F_1 &\xrightarrow{g_2} F_2 := 4w_1 + 4w_2w_3 \boxed{-4w_1w_2w_3} + 2z_1 + z_0 - (2a_1 + a_0 + 2b_1 + b_0) \\ F_2 &\xrightarrow{g_3} F_3 := 4w_1 \boxed{-4w_1w_2w_3} + 2w_2 + 2w_3 + z_0 - (2a_1 + a_0 + 2b_1 + b_0) \\ F_3 &\xrightarrow{g_4} F_4 := 4a_1b_1 - \boxed{4a_1b_1w_2w_3} + 2w_2 + 2w_3 + z_0 - (2a_1 + a_0 + 2b_1 + b_0) \\ F_4 &\xrightarrow{g_5} F_5 := \boxed{-4a_1b_1(a_1 + b_1 - 2a_1b_1)w_3} + 2w_3 + z_0 - (a_0 + b_0) \\ F_5 &\xrightarrow{g_6} F_6 := 2a_0b_0 + z_0 - (a_0 + b_0) \\ F_6 &\xrightarrow{g_7} r := 0 \end{aligned} \tag{6}$$

Please note that all variables in polynomials are Boolean, hence a term like x^n can be replaced by x. The process of dividing F by $g_1, \ldots, g_7$ is called *backward rewriting*.

The monomials in the dashed boxes in (6) are called *vanishing monomials*. These monomials appear in the intermediate steps of backward rewriting. However, they reduce to zero in the next steps. For example, the vanishing monomial $-4w_1w_4$ is generated in the first step of backward rewriting in (6). It reduces to zero after five steps of division. Recall that the early cancellation of these vanishing monomials is crucial for scaling to large non-trivial multiplier architectures.

IV. Automatic Debugging and Fixing

In this section the proposed approach for automatic debugging and fixing of gate-level arithmetic circuits is introduced. At first, we define the fault model. Then, the limitations of a pure SCA-based method for debugging and fixing are shown. Subsequently, we give an overview of the three phases of the proposed method. Finally, each phase is detailed.

A. Fault Model

In this paper we target complex gate-level arithmetic circuits with a particular focus on integer multipliers as these are known to be very hard in both design and formal verification. As a consequence, we consider *gate misplacement* as our fault model. This well known fault model changes the functionality of the design by a wrong gate [21], [16], [17]. Such faults are likely to occur, for example, when a synthesis tool makes a mistake when optimizing the circuit. Another prominent example of introducing such kind of faults is a bug in a multiplier generator tool which are used to create a dedicated multiplier architecture (under given constraints). To be somewhat more precise, when looking on the overall structure of a multiplier it can be seen that a multiplier consists of three stages, i.e. *Partial Product Generator* (PPG), *Partial Product Accumulator* (PPA), and *Final Stage Adder* (FSA). In our experiments later we will consider faults in each of these stages and the effect on debugging and fixing.

B. Limitations of SCA for Debugging

As has been shown in recent papers, SCA-based method can be used to prove the correctness of large and complex arithmetic circuits [9], [10]. However, SCA-based approaches suffer from two major limitations when employed for debugging:

1) Vanishing Monomials in Remainder: In a bug-free gate level circuit, vanishing monomials are generated during backward rewriting and they reduce to zero after some division steps. Early cancellation of vanishing monomials based on the reported logic rewriting scheme is the major reason for scaling to complex multipliers in [9], [10]. However, in a buggy arithmetic circuit, it is possible that the vanishing monomials propagate to the remainder because of a bug. To illustrate this phenomenon, we show the backward rewriting process for a buggy 2-bit adder (cf. Fig. 1b):

$$
\begin{aligned}
F &\xrightarrow{g_1} F_1 := 4w_1 + 4w_4\boxed{-4w_1w_4} + 2z_1 + z_0 - (2a_1 + a_0 + 2b_1 + b_0)\\
F_1 &\xrightarrow{g_2} F_2 := 4w_1 + 4w_2w_3\boxed{-4w_1w_2w_3} + 2z_1 + z_0 - (2a_1 + a_0 + 2b_1 + b_0)\\
F_2 &\xrightarrow{g_3} F_3 := 4w_1\boxed{-4w_1w_2w_3} + 2w_2 + 2w_3 + z_0 - (2a_1 + a_0 + 2b_1 + b_0)\\
F_3 &\xrightarrow{g_4} F_4 := 2a_1 + 2b_1 - 4a_1b_1\boxed{-4(a_1 + b_1 - a_1b_1)w_2w_3} + 2w_2 + 2w_3 + z_0\\
&\quad - (a_0 + b_0)\\
F_4 &\xrightarrow{g_5} F_5 := 4a_1 + 4b_1 - 8a_1b_1\boxed{-4(a_1 + b_1 - 2a_1b_1)w_3} + 2w_3 + z_0 - (a_0 + b_0)\\
F_5 &\xrightarrow{g_6} F_6 := 4a_1 + 4b_1 - 8a_1b_1\boxed{-4(a_1 + b_1 - 2a_1b_1)a_0b_0} + 2a_0b_0 + z_0\\
&\quad - (a_0 + b_0)\\
F_6 &\xrightarrow{g_7} r := 4a_1 + 4b_1 - 8a_1b_1\boxed{-4(a_1 + b_1 - 2a_1b_1)a_0b_0} \qquad (7)
\end{aligned}
$$

The final remainder r of backward rewriting (result of the division of F_6) is not equal to zero. Therefore, the circuit is buggy. The generated remainder consists of two parts. The first part $4a_1 + 4b_1 - 8a_1b_1 = 4 \times (a_1 + b_1 - 2a_1b_1)$ is composed of three monomials which originate from the difference in the buggy and correct gate polynomials at gate g_4 (see Fig. 1a and Fig. 1b): $P_{buggy} - P_{correct} = P_{OR} - P_{AND} = a_1 + b_1 - 2a_1b_1$. However, the second part of the remainder $-4(a_1 + b_1 - 2a_1b_1)a_0b_0$ (shown in the dashed box) is a part of vanishing monomial propagated to the remainder due to the bug presence. If we apply the approach from [16] to the just discussed example it fails. The reason is that for the vanishing monomials there will be no counterpart monomials when executing the forward rewriting and backward rewriting as presented in [16]. Furthermore, also the SCA-based method from [17] fails. This method extracts the difference polynomial per gate and compares it with the remainder. However, this is not possible if a vanishing monomial appears. Finally, note that all experiments in both papers only consider simple adder and multiplier architectures, i.e. carry save adders are used and hence no vanishing monomials occurs.

2) Blow-up during Verification of Buggy Circuits: If there is a bug close to POs of a large arithmetic circuit, a blow-up happens in the number of monomials during backward rewriting and hence the SCA-based verification method fails. The reason is that a buggy gate adds several monomials, i.e. the difference between buggy and correct gate polynomials, to the process of backward rewriting. Despite other monomials, these monomials do not cancel and grow exponentially in the subsequent steps of the division when moving towards the inputs.

To overcome these limitations, we take advantage of both SAT and SCA in our approach. An overview is presented in the next section.

C. Overview of Proposed Method

Algorithm 1 shows the pseudo-code of our proposed approach. Before we go into the details, it can be seen that our approach consists of three phases: *Verification*, *Localization*, and *Fixing*. In each phase we employ SAT and SCA for individual subtasks as both have pros and cons. We explain the underlying decisions wrt. the chosen method. We provide a summary up-front in Table I. The first column gives the name of the phase. The second column shows the subtask, if applicable. The third column distinguishes per phase/subtask between SCA and SAT. The fourth to seventh column defines whether the circuit is bug-free or not. In case of a bug, we subdivide the circuit into three regions: I, II, III which just defines the depth of the bug seen from the inputs (so III means a deep bug near to the POs). Note that *'+'* means that the respective method gives a result, and *'-'* that it fails. Finally, in the rightmost column we show the conclusion that can be drawn; which also has been finally implemented in our approach. In the following sections we describe now each phase in more detail.

D. Phase 1: Verification

The first phase of our proposed approach is verification (see Line 1 – Line 2 in Algorithm 1). For a complex gate level arithmetic circuit we want to determine whether the circuit is correct or not. As already explained in the introduction and confirmed later in the experiments in Section V, we can observe that SAT is very fast in disproving, i.e. to show that the circuit is buggy. However, SAT fails (time out) when the circuit is correct. In contrast, SCA is one of the best approaches for verifying a bug-free arithmetic circuit, especially when

TABLE I
APPLICABILITY OF SCA AND SAT IN DIFFERENT PHASES OF DEBUGGING

Phase	Subtask	Method	Bug level: Bug-free	I	II	III	Conclusion
verification		SCA	+	+	-	-	Using SAT and SCA in parallel (SAT for buggy and SCA for correct circuits)
		SAT	-	+	+	+	
Localization	Extracting Initial Suspicious Gates	SCA		+	-	-	Using SAT
		SAT		+	+	+	
	Generating Test-vectors	SCA		+	-	-	Using SAT
		SAT		+	+	+	
	Refining Suspicious Gates	SCA		++	++	++	Using SCA because it is faster
		SAT		+	+	+	
Fixing (correct fix/incorrect fix)		SCA		+/+*	+/-	+/-	Using SAT and SCA in parallel (SAT when fix does not work and SCA for proving correctness)
		SAT		-/+	-/+	-/+	

+: Applicable ++: Applicable and fast -: Not applicable
*The sign before the slash (after the slash) describes the applicability of the method when the fix at the candidate location is correct (incorrect).

Algorithm 1 Proposed method for Debugging and Fixing

Input: Arithmetic circuit C, Golden circuit C_G
Output: Correct circuit
1: VerifyWithParallelSAT_SCA (C,C_G) ▷ Verification
2: **if** C is bug-free **then return** C
3: **else**
4: $SG \leftarrow$ ExtractSGWithSAT (C, C_G) ▷ Localization
5: $i \leftarrow 0$; $v \leftarrow \emptyset$
6: **while** Size(SG) > 1 **and** $i < 10$ **do**
7: $v \leftarrow$ GenerateTestvectorWithSAT (C, C_G, v)
8: $C_X \leftarrow$ InsertXOR (C, SG, v)
9: $SG \leftarrow$ RefineSGWithSCA(C_X)
10: **if** SG has not changed **then** $i \leftarrow i + 1$
11: **else** $i \leftarrow 0$
12: **end if**
13: **end while**
14: $C_F \leftarrow$ FixWithParallelSAT_SCA (C, C_G, SG) ▷ Fixing
15: **end if**
16: **return** C_F

advanced rewriting techniques are employed, for instance XOR-rewriting and common-rewriting [9], [10].

Nevertheless, its performance is poor when there is a bug close to the POs (see previous discussion in Section IV-B). In order to take advantages of both SAT and SCA, we run them in parallel in our approach. When we obtain a result from one of the methods, we terminate the other one. As a result, buggy and bug-free circuits can be verified in acceptable time.

E. Phase 2: Localization

The second phase of our proposed method is localization (see Line 4 – Line 13 in Algorithm 1). The goal of this phase is to extract candidates for the location of the bug in the circuit. In the first step of localization, an initial list of suspicious buggy gates are extracted (Line 4). Next, a test-vector is generated for the buggy circuit (Line 7). Subsequently, XOR gates are inserted just after each suspicious gate and the test-vector is applied to the primary inputs (PIs) (Line 8). Finally, the list of suspicious gates is refined by backward rewriting and evaluating remainder (Line 9). This process continues iteratively until the size of the set of suspicious gates SG reduces to 1 or the SG does not change after 10 iterations. In following we detail each step and explain for which subtask we employ which method, i.e. SCA or SAT (remember to see also Table I for a summary).

1) Extracting Initial Suspicious Gates: Complex arithmetic circuits usually consist of many logical gates. Therefore, if there is a bug in the circuit, the size of the search space (i.e. the number of suspicious gates) will be large (number of gates). Thus, a pre-process to reduce the size of the search space is essential. To this end, the following method is proposed:

1) Use formal to identify outputs which are affected by the bug (i.e. there is an input vector such that the golden and buggy circuit differ)
2) Create cones for these outputs based on the gates which are connected
3) Determine the gates that are in the intersection of all cones; they form the initial set of suspicious gates

This task can be mapped to a MITER circuit for each output bit. In different studies (not reported) we observed that SAT performed very well and an SCA-based solution only gave results for bugs in the circuit region I, i.e. bugs near to the PIs.

Coming back to the 2-bit adder example in Fig. 1b. The only affected output is Z_2. The cone for Z_2 is $C_2 = \{g_1, g_2, g_4, g_5, g_6\}$, which is also the initial suspicious gates list. Please note that if there are more than one affected output, then the intersection of output cones creates the initial suspicious gate list.

2) Generating Test-vectors: After the verification of the arithmetic circuit, a counter-example is available from the SAT-solver (SAT-solver result from Phase 1: Verification). This counter-example can be used as the initial test-vector because it presents the input values resulting in a "wrong" output value. However, we usually need more test-vectors to localize the bug in the design. To this end, we use blocking clauses and run the SAT-solver again to obtain a new test-vector.

3) Insertion of XORs: The faulty gate in the circuit is among the gates in the set of suspicious gates SG. Assume that t is a generated test-vector which exhibits the fault. Then, the bug has been activated in the circuit by t. In other words, the faulty gate has generated a "wrong" value at its output which is the negation of the correct gate value – this assumption is valid, since we consider the gate misplacement fault model. This "wrong" value is propagated through gates in the output cone of the faulty gate, and leads to the "wrong" value at the PO(s) of the circuit. Changing the output value of the faulty gate results in the correct value at the output of the circuit. Hence, the problem can be formulated as finding gates where negating their outputs corrects the final result. As the XOR gate fulfills this property, we use it as follows: Assume that $g_1, \ldots, g_n$ are the suspicious gates, and t is a test-vector. We apply t to the primary inputs of the circuit, and insert $x_1, \ldots, x_n$ which are XOR gates just after the suspicious gates. One of the inputs of each XOR gate is connected to the output of the each suspicious gate, and the other XOR input becomes a new free input. We name all these inputs $s_1, \ldots, s_n$ and call them *selectors* in the following. The problem is now to find a *selector* by setting it to 1 (all other to 0), such that the final output of the circuit becomes correct.

Consider again the buggy 2-bit adder circuit in Fig. 1b. Recall that the initial set of suspicious gates is $\{g_1, g_2, g_4, g_5, g_6\}$. If $t_1 = 1000$ (i.e. $a_1 = 1, b_1 = 0, a_0 = 0, b_0 = 0$) is a test-vector, the new circuit after applying t_1 to the PIs and inserting XOR gates just after each suspicious gate can be seen in Fig. 1c.

4) Refining Suspicious Gates: To goal of this subtask is to refine the set of suspicious gates SG. In the following we give an SCA-based method for this subtask, since it is faster than the SAT-based formulation (empirically shown in the experiments).

Based on the given test-vector t, we recompute the specification polynomial as follows: We have now concrete input values from t which are applied to the original specification polynomial. Please note that the input of the new circuit (current problem instance) are only the selectors. The gate polynomials of the buggy 2-bit adder with the previously inserted XORs and the test-vector $t_1 = 1000$ are:

$$\begin{array}{ll} x_5 := Z_2 - s_5 - p_5 + 2s_5p_5 & x_2 := w_2 - s_2 - p_2 + 2s_2p_2 \\ g_1 := p_5 - w_1 - w_4 + w_1w_4 & x_3 := w_3 - s_3 - p_3 + 2s_3p_3 \\ x_4 := w_4 - s_4 - p_4 + 2s_4p_4 & g_4 := p_1 - 1 \\ g_2 := p_4 - w_2w_3 & g_5 := p_2 - 1 \\ g_3 := Z_1 - w_2 - w_3 + 2w_2w_3 & g_6 := p_3 - 0 \\ x_1 := w_1 - s_1 - p_1 + 2s_1p_1 & g_7 := Z_0 - 0 \end{array} \tag{8}$$

Now, backward rewriting is performed. The resulting remainder is different from 0 and only depends on the selector variables. Before showing backward rewriting for the concrete example, two important points should be noticed: 1) the terms containing $s_m \times s_n$ (i.e. multiplication of selectors) are reduced to 0 during backward rewriting, because only one of the selectors should be equal to 1; 2) due to the fact that we are dealing with a 2-bit adder, the specification polynomial and subsequently all the polynomials during backward rewriting should be modulo 2^{2+1}, because the maximum output size for addition of two n-bit numbers gives $n+1$ bits.

For our running example we get:

$$\begin{array}{l} F \xrightarrow{x_5} F_1 := 4s_5 + 4p_5 + 2z_1 + z_0 - 2 \\ F_1 \xrightarrow{g_1} F_2 := 4s_5 + 4w_1 + 4w_4 - 4w_1w_4 + 2z_1 + z_0 - 2 \\ \quad \ldots \\ F_{10} \xrightarrow{g_6} F_{11} := 4s_1 - 2s_2 - 2s_3 + 4s_5 + z_0 + 4 \\ F_{11} \xrightarrow{g_7} \boxed{r := 4s_1 - 2s_2 - 2s_3 + 4s_5 + 4} \end{array} \tag{9}$$

To correct the 2-bit adder circuit, the remainder $4s_1 - 2s_2 - 2s_3 + 4s_5 + 4$ should become 0. So, we should find all possible combinations for the selectors (one-hot encoding) such that $r = 0$. We get:

$$\begin{array}{lllll} s_1 = 1, & s_2 = 0, & s_3 = 0, & s_4 = 0, & s_5 = 0 \implies \boxed{\text{r=8 mod 8 = 0}} \\ s_1 = 0, & s_2 = 1, & s_3 = 0, & s_4 = 0, & s_5 = 0 \implies r = 2 \\ s_1 = 0, & s_2 = 0, & s_3 = 1, & s_4 = 0, & s_5 = 0 \implies r = 2 \\ s_1 = 0, & s_2 = 0, & s_3 = 0, & s_4 = 1, & s_5 = 0 \implies r = 4 \\ s_1 = 0, & s_2 = 0, & s_3 = 0, & s_4 = 0, & s_5 = 1 \implies \boxed{\text{r=8 mod 8 = 0}} \end{array} \tag{10}$$

As can be seen, when setting s_1 or s_5 to 1, the remainder becomes 0. Hence, the suspicious gates list is reduced to $\{g_4, g_1\}$ whose outputs are connected to x_1 and x_5, respectively. In the next iteration, XOR gates are inserted only after these two suspicious gates, and another test-vector is applied to PIs and suspicious gates are refined. Nevertheless, in this concrete example the suspicious gates list cannot be further reduced even after applying all test-vectors. In large arithmetic circuits, the number of test-vectors are extremely large. Therefore, in order to avoid repeating subtasks 2, 3, and 4 for all existing test-vectors, we use a termination criteria of 10 iterations for the while-loop in Algorithm 1. In other words, if the size of suspicious gates list does not change after 10 iterations, then the suspicious gates list is sent to fixing phase.

F. Phase 3: Fixing

The final phase of our proposed method is Fixing (see Line 14 – Line 16 in Algorithm 1). Based on the extracted candidates in the bug localization phase, we can create a list of potential gate replacements. For example, if we assume that the used library for creating arithmetic circuits consists of the basic logical gates $\{AND, OR, XOR, NOT\}$, then there are two possible gate replacements for each candidate. To find the correct gate replacement, we first choose one of the changes from the list, and apply it to the circuit. Then, we perform parallel verification using SCA and SAT (see Section IV-D). Therefore, after gate replacement, if the circuit is still buggy the SAT-based verification returns a counter-example and we continue with the next possible replacement. Otherwise, if the circuit can be fixed with the current gate replacement, the SCA-based verification successfully proves this.

Considering again the running 2-bit buggy adder example of Fig. 1b, from the localization phase we know that g_4 and g_1 are the final suspicious gates. The corresponding list of gate replacements which may fix the bug is therefore $\{g_4(OR) \rightarrow g_4(XOR), \; g_4(OR) \rightarrow g_4(AND), \; g_1(OR) \rightarrow g_1(XOR), \; g_1(OR) \rightarrow g_1(AND)\}$. First, g_4 is converted to an XOR gate, and the circuit is verified. Because the circuit is still buggy, a counter-example is returned. When applying the second change and verifying the circuit, the final remainder of SCA-based verification becomes 0 and hence we have found the fix.

V. Experimental Results

We have implemented our approach in C++. For SCA we implemented Gröbner basis reduction including the rewriting techniques proposed in [9]. The experiments have been carried out on an Intel(R) Core(TM) i5-4300M CPU 2.60 GHz with 16 GByte of main memory. In order to evaluate the efficiency of our combined SCA and SAT approach, we consider different complex multiplier architectures generated by the *Arithmetic Module Generator* [22]. These multipliers are in the form of RTL Verilog code. Thus, we run *Yosys* [23] (commands: *read_verilog; proc; opt; write_verilog*) to synthesize them to a gate-level netlist. The generated multipliers consist of three stages: 1st) *Partial Product Generator* (PPG), 2nd) *Partial Product Accumulator* (PPA) which is a multi-operand adder, and 3rd) *Final Stage Adder* (FSA) which is a two-operand adder. All the benchmarks are named based on the type of the used architecture in the different stages (this includes for instance Booth encoding and Carry look-ahead adders in the respective stages); see details in the legend below the result table later. We also used MiniSat v1.14 [24] for SAT-solving in our experiments.

In Table II, we report the results of applying debugging methods to different types of multipliers. Please note that the *Time-Out* (TO) has been set to 24 hours. The first column of Table II shows the type of the multiplier (see below the table for the abbreviations). The second column *I/O bits* gives the number of input and output bits. The third column *Bug* lists whether the circuit is bug-free, or the stage where the bug has been inserted randomly.

The results of the verification phase are reported in the fourth column **Verification**, which consists of the five following subcolumns: While *SCA* and *SAT* refer to our SCA-implementation and MiniSat, respectively, *Comm.* reports the results of the commercial formal verification tool OneSpin. Next, our integrated approach is given in subcolumn *Ours*, and finally *Imp.* presents the improvement of our approach compared to the commercial tool. As can be seen pure SCA-based verification only works when there is no bug in the circuit or the bug is in the first stage (i.e. PPG) of the design. In contrast, pure SAT-based verification times-out for bug-free circuits already for the multipliers with only 16/32 I/O bits. The commercial tool also times-out for bug-free multipliers bigger than 16/32 I/O bits. In contrast, our integrated verification method (Section IV-D) can verify bug-free multipliers, and also buggy circuits when the bug is in any stage of the design. Our verification method is up to 641 times faster than the commercial tool.

TABLE II
RESULTS OF DEBUGGING DIFFERENT TYPES OF MULTIPLIERS (RUN-TIMES IN SECONDS)

Benchmark post-synth.	I/O	Bug	Verification SCA	SAT	Comm.	Ours	Imp.	Localization SAT [11]	Ours	Imp.	Fixing SCA	SAT	Ours	Overall Ours	SOTA [16]	[17]
BP-CT-BK	16/32*	Bug-free	0.29	TO	218.00	0.34	641.18x							0.3		
		Stage 1	0.31	0.03	0.06	0.03	2.00x	13.1	5.0	2.62x	0.66	TO	0.44	5.5	F	F
		Stage 2	TO	0.02	0.05	0.02	2.50x	14.9	3.7	4.03x	TO	TO	0.41	4.1	F	F
		Stage 3	TO	0.03	0.07	0.03	2.33x	10.7	3.6	2.97x	TO	TO	0.46	4.1	F	F
SP-WT-CL	32/64	Bug-free	9.09	TO	TO	10.70	-							10.7		
		Stage 1	10.42	0.20	0.38	0.22	1.73x	115.6	61.5	1.88x	19.51	TO	10.95	72.7	F	F
		Stage 2	TO	0.16	0.41	0.20	2.05x	1208.0	108.5	11.13x	TO	TO	11.10	119.8	F	F
		Stage 3	TO	0.12	0.35	0.14	2.50x	165.4	40.1	4.12x	TO	TO	10.87	51.1	F	F
BP-AR-RC	32/64	Bug-free	3.79	TO	TO	4.54	-							4.5		
		Stage 1	4.37	0.14	0.24	0.15	1.60x	208.7	37.7	5.54x	8.29	TO	4.63	42.5	F	F
		Stage 2	TO	0.10	0.31	0.13	2.38x	89.3	25.7	3.47x	TO	TO	4.93	30.8	F	F
		Stage 3	TO	0.08	0.30	0.09	3.33x	60.8	24.7	2.46x	TO	TO	4.81	29.6	F	F
BP-WT-CL	32/64	Bug-free	11.59	TO	TO	13.84	-							13.8		
		Stage 1	12.51	0.08	0.23	0.08	2.87x	291.2	67.5	4.31x	24.23	TO	13.93	81.5	F	F
		Stage 2	TO	0.11	0.27	0.12	2.25x	228.2	105.3	2.17x	TO	TO	14.20	119.6	F	F
		Stage 3	TO	0.45	0.32	0.56	0.57x	148.1	43.7	3.39x	TO	TO	15.66	59.9	F	F
SP-WT-CL	64/128	Bug-free	161.78	TO	TO	192.14	-							192.1		
		Stage 1	184.30	1.14	12.66	1.22	10.38x	1107.4	343.3	3.23x	348.48	TO	193.52	538.0	F	F
		Stage 2	TO	3.86	77.00	4.14	18.60x	1745.6	250.6	6.97x	TO	TO	212.84	467.6	F	F
		Stage 3	TO	0.57	4.46	0.63	7.08x	1047.9	518.5	2.02x	TO	TO	192.78	711.9	F	F
SP-CT-BK	64/128	Bug-free	67.04	TO	TO	79.24	-							79.2		
		Stage 1	73.30	1.79	42.47	1.91	22.24x	1192.3	704.3	1.69x	140.94	TO	81.25	787.5	F	F
		Stage 2	TO	0.42	22.05	0.46	47.93x	677.7	265.6	2.55x	TO	TO	79.73	345.8	F	F
		Stage 3	TO	0.88	9.56	0.94	10.17x	3612.0	349.8	10.33x	TO	TO	80.31	431.1	F	F
BP-WT-CL	64/128	Bug-free	226.65	TO	TO	271.93	-							271.9		
		Stage 1	255.82	0.94	2.05	0.99	2.07x	1323.1	388.0	3.41x	489.47	TO	274.90	663.9	F	F
		Stage 2	TO	0.43	2.61	0.53	4.92x	1669.0	514.5	3.24x	TO	TO	272.99	788.0	F	F
		Stage 3	TO	2.28	14.85	2.40	6.19x	698.6	241.1	2.90x	TO	TO	274.44	517.9	F	F

Stage 1 ⇒ **SP**: Simple partial product generator **BP**: Booth partial product generator TO: Time-Out F: Failed
Stage 2 ⇒ **AR**: Array **WT**: Wallace tree **CT**: Compressor tree *Due to page limitation we only report results
Stage 3 ⇒ **RC**: Ripple carry adder **CL**: Carry look-ahead adder **BK**: Brent-kung adder for one 16/32 I/O bits complex multiplier.

The fifth column **Localization** shows the run-times of the localization phase. We have compared our method against SAT-based localization [11]. While SAT-based localization is able to compute the set of fault candidates for all three stages, our method is faster on all benchmarks. The respective improvement is listed in the third subcolumn. As can be seen we achieve improvements of up to a factor of 11.

The experimental results for the fixing phase are reported in the sixth column **Fixing**. As can be seen, the pure SCA-based fixing method is only able to fix bugs in the first stage (PPG) of the multiplier. SAT-based fixing fails for all the cases because when a correct gate replacement is considered, this method cannot verify the bug-free circuit. The experimental results confirm that our fixing method can fix the bugs in any stage of the design.

The overall run-time of our proposed method is reported in the seventh column **Overall**. It gives the sum of the run-times of each phase, i.e. verification, localization, and fixing.

The eighth column shows the results using the *State-Of-The-Art* (SOTA) SCA-based methods [16] and [17]. However, both methods fail to debug the considered complex multipliers due to appearance of vanishing monomials in the final remainder (see discussion and example in Section IV-B).

A final remark on the relevance of vanishing monomials for the considered multiplier architectures: On average when running our proposed approach for the benchmarks 135,498 vanishing monomials have been canceled during the divisions. This number has been calculated running the complete flow, i.e. all three phases per benchmark.

VI. CONCLUSION

In this paper, we have introduced a novel approach based on the combination of SCA and SAT for automatic debugging and fixing complex arithmetic circuits. The proposed approach consists of three phases. First, the arithmetic circuit is verified. Then, a list of candidates for the location of the bug is extracted. Finally, the design is fixed. The experimental results showed that our approach allows for debugging and fixing of complex arithmetic circuits while other state-of-the-arts methods fail.

REFERENCES

[1] D. Stoffel and W. Kunz, "Equivalence checking of arithmetic circuits on the arithmetic bit level," *TCAD*, vol. 23, no. 5, pp. 586–597, 2004.
[2] E. Pavlenko, M. Wedler, D. Stoffel, O. Wienand, E. Karibaev, and W. Kunz, "Modeling of custom-designed arithmetic components in ABL normalization," in *FDL*, 2008, pp. 124–129.
[3] S. Vasudevan, V. Viswanath, R. W. Sumners, and J. A. Abraham, "Automatic verification of arithmetic circuits in rtl using stepwise refinement of term rewriting systems," *TC*, vol. 56, no. 10, pp. 1401–1414, 2007.
[4] D. Kapur and M. Subramaniam, "Mechanical verification of adder circuits using rewrite rule laboratory," *Formal Methods in System Design: An International Journal*, vol. 13, no. 2, pp. 127–158, 1998.
[5] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner basis reductions for formal verification of Galois field multipliers," in *DATE*, 2012, pp. 899–904.
[6] ——, "Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits," *TCAD*, vol. 32, no. 9, pp. 1409–1420, Sept 2013.
[7] 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.
[8] M. Ciesielski, C. Yu, D. Liu, W. Brown, and A. Rossi, "Verification of gate-level arithmetic circuits by function extraction," in *DAC*, 2015, pp. 52:1–52:6.
[9] 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.
[10] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD*, 2017.
[11] A. Smith, A. G. Veneris, and A. Viglas, "Design diagnosis using boolean satisfiability," in *ASP-DAC*, 2004, pp. 218–223.
[12] B. Le, H. Mangassarian, B. Keng, and A. G. Veneris, "Non-solution implications using reverse domination in a modern SAT-based debugging environment," in *DATE*, 2012, pp. 629–634.
[13] B. Keng and A. G. Veneris, "Path-directed abstraction and refinement for SAT-based design debugging," *TCAD*, vol. 32, no. 10, pp. 1609–1622, 2013.
[14] A. M. Gharehbaghi and M. Fujita, "A new approach for debugging logic circuits without explicitly debugging their functionality," in *ATS*, 2016, pp. 31–36.
[15] H. Riener and G. Fey, "Exact diagnosis using boolean satisfiability," in *ICCAD*, 2016.
[16] S. Ghandali, C. Yu, D. Liu, W. Brown, and M. Ciesielski, "Logic debugging of arithmetic circuits," in *ISVLSI*, 2015, pp. 113–118.
[17] F. Farahmandi and P. Mishra, "Automated test generation for debugging arithmetic circuits," in *DATE*, 2016, pp. 1351–1356.
[18] ——, "Automated debugging of arithmetic circuits using incremental gröbner basis reduction," in *ICCD*, 2017, pp. 1–6.
[19] D. A. Cox, J. Little, and D. O'Shea, *Ideals Varieties and Algorithms*. Springer, 1997.
[20] W. W. Adams and P. Loustaunau, *An Introduction to Grobner Bases*. American Mathematical Society, 1994.
[21] A. Veneris and I. N. Hajj, "Design error diagnosis and correction via test vector simulation," *TCAD*, vol. 18, no. 12, pp. 1803–1816, 1999.
[22] "Arithmetic module generator based on acg," available at www.aoki.ecei.tohoku.ac.jp/arith/, 2015.
[23] C. Wolf, "Yosys open synthesis suit," available at http://www.clifford.at/yosys/, 2015.
[24] N. Eén and N. Sörensson, "Minisat," available at http://minisat.se/, 2008.