Towards Polynomial Formal Verification of Complex Arithmetic Circuits

Rolf Drechsler[1,2] Alireza Mahzoon[1] Mehran Goli[1,2]
[1]Institute of Computer Science, University of Bremen, Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{drechsle, mahzoon, mehran}@informatik.uni-bremen.de

***Abstract*—With the growing demands for highly area-efficient, delay-optimized, and low-power designs, the complexity of digital circuits is increasing as well. Especially, a wide variety of arithmetic circuits, including different types of adders, multipliers, and dividers have been proposed to meet the demands in applications such as cryptography and Artificial Intelligence (AI). Some of these arithmetic circuits have highly parallel architectures and contain millions of gates; as a result, they are extremely error-prone. In the last 30 years, several formal verification methods have been proposed to verify arithmetic circuits. These methods report very good results when it comes to the verification of adders and structurally simple multipliers. Moreover, their space and time complexities are polynomial, i.e, they are scalable. However, when it comes to the verification of structurally complex multipliers, the story is different.**

In this paper, we investigate the space and time complexity of verifying a structurally complex multiplier using a word-level verification method. We prove that the space and time complexity is always exponential. Then, we introduce a new verification strategy that takes advantage of several verification engines. We show that the polynomial formal verification of the complex multiplier becomes possible if the correctness of each stage is verified using the proper verification method. Our verification strategy can be applied to other complex digital circuits.

I. Introduction

In the last 30 years, the verification community has achieved many successes in formal verification of a wide variety of digital designs, particularly arithmetic circuits. Several formal verification methods have been proposed to ensure the correctness of highly complex and big arithmetic blocks, including adders, multipliers, and dividers. The bit-level methods, including the techniques based on *Binary Decision Diagrams* (BDDs) [1] and *Boolean Satisfiability* (SAT) [2], [3], report very good results for the verification of adders and subtractors. On the other hand, the word-level methods, including the approaches based on *Symbolic Computer Algebra* (SCA) [4]–[9] and *Binary Moment Diagram* (*BMD and K*BMD) [10], are used to verify multipliers and dividers. Despite this huge progress, the space and time complexity of these verification techniques are not fully investigated. As a result, there is always an unpredictability in the performance of a method when it comes to the verification of different architectures.

The unpredictability in the performance of a verification method has some consequences: 1) It is not clear whether the method is scalable for the verification of a specific architecture or a group of architectures. For example, it has not been proven whether SAT-based verification is scalable for parallel prefix adders. 2) It is not possible to predict the required resources for the verification process, i.e., the run-time and memory usage cannot be estimated. 3) It is impossible to compare two available verification techniques and choose the best one with respect to the architecture. In order to alleviate these issues, the time and space complexity of formal verification methods has to be calculated. In particular, we are interested in polynomial formal verification, where the space and time complexity is bounded by $O(n^m)$ (n is one of the circuit's characteristics, such as the number of input bits, and m is a positive number).

Recently, researchers made some efforts to calculate the space and time complexity of both bit-level (e.g., BDD) and word-level (e.g., SCA and *BMD) verification methods. PolyAdd [11] proves that the formal verification of three adder architectures is possible in polynomial time using BDDs. The proof is based on the fact that underlying BDDs remain polynomial during the whole construction process. However, PolyAdd does not calculate the upper-bound complexities. Authors of [12], [13] extend PolyAdd by obtaining the verification time complexities for a ripple carry adder and a conditional sum adder, respectively. The method in [14] proves that polynomial formal verification of three prefix adders (i.e., serial prefix adder, Ladner-Fischer adder, and Kogge-Stone adder) is possible using BDDs. Authors of [15] and [16] show that polynomial formal verification of structurally simple multipliers is possible using *BMDs and SCA.

The polynomial formal verification of structurally complex multipliers is a challenging task that has not been studied yet. The verification methods such as RevSCA-2.0 [7] reports good results for the structurally complex multipliers in practice. However, it is impossible to prove their polynomial complexity for all multiplier architectures due to some heuristics in their flow (e.g. dynamic backward rewriting). In this paper, we come up with a new hybrid verification method that takes advantage of both SCA and BDD to verify a structurally complex multiplier in polynomial time. We first prove that a pure SCA-based verification method has an exponential complexity for a complex multiplier. Then, we introduce our hybrid verification technique consisting of three main phases: 1) replacing the final stage adder with a ripple carry adder, 2) SCA-based verification of the structurally simple multiplier after the replacement, and 3) BDD-based verification of the final stage adder. Our method is applicable to structurally complex multipliers where the boundaries between stages are available. We theoretically prove that both verification phases (i.e., BDD and SCA-based verification) can be carried out in polynomial time, meaning that our proposed method is scalable. We also evaluate our proposed method experimentally with various multiplier architectures. The experimental results confirm the efficiency of our proposed method in practice.

This work was supported by DFG within the Reinhart Koselleck Project *PolyVer* (DR 287/36-1).

978-1-6654-9431-1/22/$31.00 ©2022 IEEE

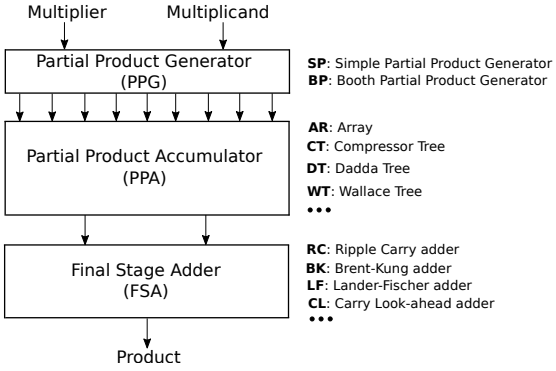

Fig. 1: General multiplier structure

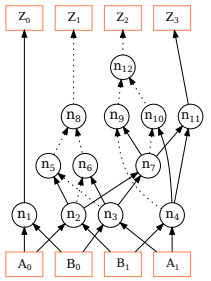

Fig. 2: 2×2 mult

$$SP := 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - A \times B$$
$$SP \xrightarrow{P_{Z_3}} SP_1 := 8n_{11} + 4Z_2 + 2Z_1 + Z_0 - A \times B$$
$$SP_1 \xrightarrow{P_{Z_2}} SP_2 := 8n_{11} + 4 - 4n_{12} + 2Z_1 + Z_0 - A \times B$$
$$SP_2 \xrightarrow{P_{n_{12}}} SP_3 := 8n_{11} + 4n_9 + 4n_{10} - 4n_9n_{10} + 2Z_1 + Z_0 - A \times B$$
$$SP_3 \xrightarrow{P_{n_{11}}} SP_4 := 8n_4n_7 + 4n_9 + 4n_{10} - 4n_9n_{10} + 2Z_1 + Z_0 - A \times B$$
$$\vdots$$
$$SP_{13} \xrightarrow{P_{n_3}} SP_{14} := n_1 + n_2 - (A_0B_0 + A_0B_1)$$
$$SP_{14} \xrightarrow{P_{n_2}} SP_{15} := n_1 - (A_0B_0)$$
$$SP_{15} \xrightarrow{P_{n_1}} r := 0$$

Fig. 3: Backward rewriting steps

II. Preliminaries

A. Multiplier Structure

Fig. 1 shows an integer multiplier consisting of three stages: (1) *Partial Product Generator* (PPG) which generates partial products from two inputs, (2) *Partial Product Accumulator* (PPA) which reduces partial products using multi-operand adders and computes their sums, and (3) *Final Stage Adder* (FSA) which converts these sums to the corresponding binary output.

There are several architectures for each stage of a multiplier. These architectures are chosen with respect to the design goals, e.g. small area, low delay, and the small number of wiring tracks. If the second and third stages of a multiplier are made of only atomic blocks (e.g. half-adders, full-adder, and 4:2 compressors) [5], we call it structurally simple. The SCA-based verification reports very good results for the structurally simple multipliers. On the other hand, if the second and third stages (usually FSA) are not fully made of atomic blocks, we call them structurally complex. Formal verification of structurally complex multipliers is a challenge for the SCA-based methods. Thus, researchers usually take advantage of some heuristics in the flow to make the SCA-based verification possible.

B. Verification using SCA

We briefly summarize some basics of SCA:

- **Monomial:** power product of the variables, i.e. $M = x_1^{a_1} x_2^{a_2} \dots x_n^{a_n}$ where $a_i \geq 0$.
- **Polynomial:** finite sum of monomials, i.e. $P = c_1 M_1 + \dots + c_j M_j$ with coefficients in field k.
- **Division:** Assuming p is a polynomial and F is a set of polynomials, the division of p by F is denoted by $p \xrightarrow{F} r$, where r is called remainder.

The goal of SCA-based verification is to formally prove that all signal assignments consistent with the gate-level or *AND Inverter Graph* (AIG) representation evaluate the *Specification Polynomial* (SP) to 0. The SP determines the function of an arithmetic circuit based on its inputs and outputs, e.g. for the 2×2 multiplier of Fig. 2 $SP = 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (2A_1 + A_0)(2B_1 + B_0)$, where $8Z_3 + 4Z_2 + 2Z_1 + Z_0$ represents the word-level representation of the 4-bit output, and $(2A_1 + A_0)(2B_1 + B_0)$ represents the product of the 2-bit inputs.

Before verification, the nodes of an AIG (or gates of a gate-level representation) should be modeled as polynomials describing the relation between inputs and outputs. Based on the type of nodes and edges, five different operations might happen in an AIG. Assuming z is the output, and n_i and n_j are the inputs of a node:

$$\begin{aligned} &z = n_i \Rightarrow p_N := z - n_i, && z = n_i \wedge n_j \Rightarrow p_N := z - n_i n_j, \\ &z = \neg n_i \Rightarrow p_N := z - 1 + n_i, && z = \neg n_i \wedge n_j \Rightarrow p_N := z - n_j + n_i n_j, \\ &z = \neg n_i \wedge \neg n_j \Rightarrow p_N := z - 1 + n_i + n_j - n_i n_j. && \end{aligned} \quad (1)$$

The extracted node polynomials are in the form $P_N = x - tail(P_N)$ where x is the node's output, and $tail(P_N)$ is a function based on the node's inputs. Similarly, the polynomials for the gates can be extracted in a gate-level representation (see [4], [17]).

Based on the Gröbner basis theory, all signal assignments consistent with the AIG evaluate the specification polynomial SP to 0, iff the remainder of dividing SP by the AIG node polynomials is equal to 0 (see [8] for more details).

The step-wise division of SP by node polynomials is shown in Fig. 3 for the 2×2 multiplier. Since the remainder is zero, the circuit is bug-free. In arithmetic circuits, dividing SP_i by a node polynomial $P_{N_i} = x_i - tail(P_{N_i})$ is equivalent to substituting x_i with $tail(P_{N_i})$ in SP_i. For example, dividing SP_3 by $P_{n_{11}}$ in Fig. 3 is equivalent to substituting n_{11} with $tail(P_{n_{11}}) = n_4 n_7$ in SP_3. In the results, we always replace powers $x_i^{a_i}$ with $a_i > 1$ by x_i, since x_i can only take values from $\{0, 1\}$. In the theory, this corresponds to adding $x_i^2 - x_i$ to the node polynomials. The process of step-wise division (substitution) is called *backward rewriting*. We refer to this intermediate polynomial as SP_i in the rest of the paper.

C. Verification using BDDs

We briefly summarize some basics of BDD:

- **Binary Decision Diagram (BDD):** a directed, acyclic graph whose nodes have two edges associated with the values of the variables 0 and 1. A BDD contains two terminal nodes (leaves) that are associated with the values of the function 0 or 1.
- **Ordered BDD (OBDD):** a BDD, where the variables occur in the same order in each path from the root to a leaf.
- **Reduced OBDD (ROBDD):** an OBDD that contains a minimum number of nodes for a given variable order.

We refer to ROBDD as BDD in the rest of the paper since it is the canonical representation that is used in the verification of arithmetic circuits.

The ITE operator (If-Then-Else) is used to calculate the results of the logic operations in BDDs:

$$ITE(f, g, h) = (f \wedge g) \vee (\bar{f} \wedge h), \quad (2)$$

Algorithm 1 If-Then-Else (ITE)

Input: f, g, h BDDs
Output: ITE BDD
1: **if** terminal case **then**
2: **return** $result$
3: **else if** computed-table has entry $\{f, g, h\}$ **then**
4: **return** $result$
5: **else** ▷ General case
6: $v =$ top variable for f, g, or h
7: $t = ITE(f_{v=1}, g_{v=1}, h_{v=1})$
8: $e = ITE(f_{v=0}, g_{v=0}, h_{v=0})$
9: $r = FindOrAddUniqueTable(v, t, e)$
10: $InsertComputedTable(\{f, g, h\}, r)$
11: **return** R

The basic binary operations can be presented using the ITE operator:

$$f \wedge g = ITE(f, g, 0), \qquad f \vee g = ITE(f, 1, g),$$
$$f \oplus g = ITE(f, \overline{g}, g), \qquad \overline{f} = ITE(f, 0, 1). \tag{3}$$

ITE can be also used recursively in order to compute the results:

$$ITE(f, g, h) = ITE(x_i, ITE(f_{x_i}, g_{x_i}, h_{x_i}), ITE(f_{\overline{x}_i}, g_{\overline{x}_i}, h_{\overline{x}_i})), \tag{4}$$

where f_{x_i} ($f_{\overline{x}_i}$) is the positive (negative) cofactor of f with respect to x_i, i.e., the result of replacing x_i by the value 1 (0).

The algorithm for calculating ITE operations is presented in Algorithm 1. The result is computed recursively based on Eq. (4) in this algorithm. When calculating the results of ITE operations for the f, g, h BDDs, the arguments for subsequent calls to the ITE subroutine are the subdiagrams of f, g and h. The algorithm employs two major data structures: a *Unique Table* to guarantee the canonicity of the BDDs (see Line 9), and a *Computed Table* to store results of previous computations and avoid repetition (see Line 10). The number of subdiagrams in a BDD is equivalent to the number of nodes. For each of the three arguments, the sub-routine is called at most once. Assuming that the search in the *Unique Table* is performed at a constant time, the computational complexity of the ITE algorithm, even in the worst-case, does not exceed $O(|f| \cdot |g| \cdot |h|)$, where $|f|$, $|g|$ and $|h|$ denote the size of the BDDs in terms of the number of nodes [18].

In order to formally verify an adder, we need to have the BDD representation of the outputs. Symbolic simulation helps us to obtain the BDD for each primary output. In a simulation, an input pattern is applied to a circuit, and the resulting output values are observed to see whether they match the expected values. On the other hand, symbolic simulation verifies a set of scalar tests (which usually covers the whole input space) with a single symbolic test. Symbolic simulation using BDDs is done by generating corresponding BDDs for the input signals. Then, starting from primary inputs, the BDD for the output of a gate (or a building block) is obtained using the ITE algorithm. This process continues until we reach the primary outputs. Finally, the output BDDs are evaluated to see whether they match the BDDs of an adder.

III. Verification Complexity

In this section, we first investigate the time and space complexity of backward rewriting. Then, we consider a structurally complex multiplier and prove that it has an exponential verification complexity using SCA.

A. Backward Rewriting

The main operation during backward rewriting is the substitution of gates/atomic blocks polynomials in SP_i. In order to calculate the time complexity of the whole backward rewriting process, we first have to obtain the complexity of a single substitution step. Assume that in step i of backward rewriting, the variable v is substituted by polynomial f in SP_i. The detailed substitution steps are as follows:

1) SP_i is searched for all occurrences of variable v,
2) all occurrences of variable v are substituted by f,
3) the multiplications between f and the monomials are performed and the newly generated monomials are added to SP_i, and
4) it is checked whether the newly generated monomials can be simplified with the existing monomials; if yes, the coefficients are updated.

The time complexity of a single substitution is calculated by adding up the computational complexity of each step. The computational complexity of steps 1 and 4 are dependent on the size of SP_i before and after substitution since the polynomial has to be traversed in both cases. On the other hand, the computational complexity of steps 2 and 4 relies on the number of variable v occurrences in SP_i. Finally, the backward rewriting time complexity is obtained by adding up the time complexity of each substitution step (i.e., including 4 sub-steps).

The space complexity of SCA-based verification is calculated based on the maximum size of SP_i during backward rewiring with respect to the number of variables.

B. Verification of a Structurally Complex Multiplier

Formal verification of structurally complex multipliers is a big challenge for the SCA-based verification methods. A large number of monomials are generated during the backward rewriting, leading to a memory shortage and consequently the verification failure. It has been shown experimentally in [7] that a large number of monomials (and variables) are generated during the backward rewriting of the FSA stage when it is not only made of atomic blocks (i.e., half-adders, full-adders, and 4:2 compressors). The only FSA architecture which is fully made of atomic blocks is ripple carry adder. Thus, employing other architectures (e.g., carry look-ahead adder and prefix adders) as FSA results in a structurally complex multiplier and therefore an explosion during backward rewriting.

Despite several experiments, the explosion during the verification of structurally complex multipliers has not been yet theoretically investigated. Thus, we now consider a structurally complex multiplier with a carry look-ahead adder as its FSA and prove that its verification complexity is exponential.

The Boolean formulation of a n-bit carry look-ahead adder is as follows:

$$\begin{aligned} G_i &= x_i \wedge y_i, \\ P_i &= x_i \oplus y_i, \\ c_1 &= G_0 \vee (c_0 \wedge P_0), \\ c_2 &= G_1 \vee (G_0 \wedge P_1) \vee (c_0 \wedge P_0 \wedge P_1), \\ c_3 &= G_2 \vee (G_1 \wedge P_2) \vee (G_0 \wedge P_1 \wedge P_2) \vee (c_0 \wedge P_0 \wedge P_1 \wedge P_2), \\ &\ldots, \end{aligned} \tag{5}$$

where x_i and y_i are i^{th} bits of the first and second inputs, and c_i is a carry. In a carry look-ahead adder, the carry

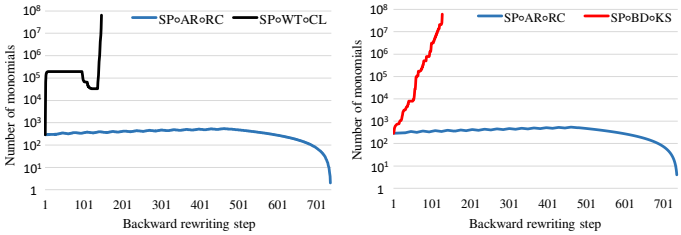

Fig. 4: SP_i size during the verification of 16×16 multipliers (**SP**: simple PPG, **AR**: array, **WT**: Wallace tree, **BD**: balanced delay tree, **RC**: ripple carry adder, **CL**: carry look-ahead adder, **KS**: Kogge-Stone adder)

bits are generated in parallel and independently. In order to compute c_i, first, the AND operations are performed, and then the consecutive OR gates (OR gates chains) are used to obtain the carry. We consider the backward rewriting for c_n which contains n consecutive OR gates. Starting from the output of the FSA, the OR gates are first visited, and their polynomials are substituted in SP_i. The polynomial for c_n after the substitution of OR gate polynomials is as follows:

$$\begin{aligned}
&\xrightarrow{c_n = X \vee Y} c_n = X + Y - XY,\\
&\xrightarrow{X = X_1 \vee Y_1} c_n = X_1 + Y_1 - X_1Y_1 + Y - X_1Y - Y_1Y + X_1Y_1Y,\\
&\xrightarrow{X_1 = X_2 \vee Y_2} c_n = X_2 + Y_2 - X_2Y_2 + Y_1 - X_2Y_1 - Y_2Y_1\\
&+ X_2Y_2Y_1 + Y - X_2Y - Y_2Y + X_2Y_2Y - Y_1Y + X_2Y_1Y\\
&+ Y_2Y_1Y - X_2Y_2Y_1Y,\\
&\ldots.
\end{aligned} \tag{6}$$

The number of variables substituted in each step of backward rewriting is growing exponentially, i.e., X, X_1, and X_2 occur 2, 4, and 8 times, respectively. As a result, the polynomial size increases exponentially in terms of the number of variables. On the other hand, the computational complexity of each substitution step depends on the number of variables. Thus, we can conclude that both the space and time complexities of the SCA-based verification are exponential.

Our proof indicates that if a structurally complex multiplier has a carry look-ahead adder as its FSA, independent of the first and second stage architectures, the verification complexity is exponential. It is possible to prove the exponential complexity for the verification of other structurally complex multipliers, containing optimized adders.

In addition to the theoretical proofs, we also confirm the exponential verification complexity of structurally complex multipliers in practice. Fig. 4 shows the size of intermediate polynomial SP_i during the verification of two 16 structurally complex multipliers, i.e. $SP \circ WT \circ CL$ and $SP \circ DT \circ BK$ against a structurally simple multiplier, i.e. $SP \circ AR \circ RC$ (see Fig. 1 for abbreviations). The complex multipliers take advantage of optimized adders in their last stage. During the backward rewriting of complex multipliers, the size of SP_i grows very fast in the first steps, and an explosion happens in the number of monomials (variables). On the other hand, the size of SP_i remains almost constant for the structurally simple multiplier until the final steps; then, it starts to decrease. As a result, the experimental results also confirm that the SCA-based verification of structurally complex multipliers has exponential complexity.

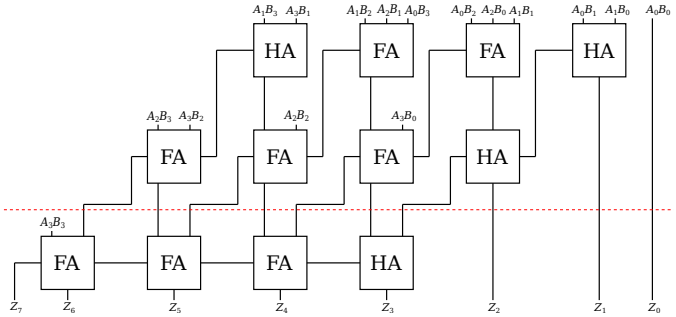

Fig. 5: A 4×4 structurally simple multiplier

Recently, researchers have extended SCA-based verification methods to verify structurally complex multipliers. For example, RevSCA-2.0 takes advantage of atomic blocks identification, vanishing monomials removal, and dynamic backward rewriting to avoid the explosion [7]. Despite the practical success of RevSCA-2.0, it is impossible to prove its polynomial space and time complexity due to the use of heuristics in its flow. Thus, we propose a new hybrid method in the next section to make the polynomial formal verification possible.

IV. Polynomial Verification using SCA and BDD

The state-of-the-art SCA-based verification methods such as RevSCA-2.0 do not need any information on the structure of multipliers in order to prove their correctness. However, they cannot guarantee the polynomial space and time complexity. Thus, they cannot be used in applications where polynomial verification has to be ensured.

On the other hand, if the design hierarchy including the boundaries between the three stages (i.e. PPG, PPA, and FSA) and the components in each stage are available, we can propose a polynomial formal verification method based on SCA and BDD. Our method consists of three main steps:

1) the final stage of the multiplier, i.e. FSA, is replaced with a ripple carry adder,
2) the new multiplier architecture is verified using SCA,
3) the FSA is verified using BDDs.

If both verification methods ensure correctness, the multiplier is bug-free. Otherwise, it is buggy.

It is now possible to calculate the space and time complexity of SCA and BDD-based methods separately and prove that they are polynomial with respect to the multiplier size.

V. Polynomial Verification of Simple Multiplier

After replacing the FSA with a ripple carry adder, the complex multiplier is now converted into a structurally simple multiplier since the second and third stages are only made of atomic blocks (e.g. half-adders and full-adders). We now calculate the space and time complexity of SCA-based verification when it comes to structurally simple multipliers. Moreover, we prove that the polynomial formal verification is possible for these circuits.

Please consider the substitution steps in Section III-A again. During backward rewriting of structurally simple multipliers, there is always one occurrence of variable v in each step. Moreover, the size of polynomial f, and consequently the number of multiplications are constant. Therefore, steps 2 and 3 of substitution have constant computational complexity. On

the other hand, for finding variable v in SP_i in step 1 and simplifying the newly generated monomials in step 4, we have to go through all variables in SP_i. Hence, the complexity of steps 1 and 4 depends on the size of SP_i with respect to the number of variables. We conclude that the time complexity of each substation step relies on the size of the current polynomial SP_i. The overall time complexity of backward rewriting is obtained by adding up the complexity of each step.

Fig. 5 shows the structure of a 4×4 simple multiplier. The second and third stages of the multiplier are only made of atomic blocks. There is always a simple word-level function that describes the relationship between inputs and outputs of an atomic block. The word-level functions for a half-adder, a full-adder, and a 4:2 compressor are as follows:

$$\begin{aligned} HA(in:X,Y \quad out:C,S) &\Rightarrow 2C+S=X+Y \\ FA(in:X,Y,Z \quad out:C,S) &\Rightarrow 2C+S=X+Y+Z \\ CM(in:X,Y,Z,W,C_{in} \quad out:Co,C,S) &\Rightarrow 2Co+2C+S=X+Y \\ &\quad +Z+W+C_{in} \end{aligned} \tag{7}$$

Eq. (7) shows that if during backward rewriting, the outputs of an atomic block are substituted by their polynomials, the size of SP_i increases by k, where $k \leqslant 2$. For an $n \times n$ multipliers, the number of atomic blocks in the second and third stages are of $O(n^2)$ complexity. Similarly, the size of specification polynomial (i.e. $SP := Z - A \times B$) is of $O(n^2)$ complexity. After substitution of atomic blocks in the second and third stages, the size of SP_i increases by $O(n^2)$.

The first stage of a multiplier with the simple partial product generator architecture is made of n^2 AND gates. Substitution of AND gates polynomials reduces the size of SP_i by 4 since a cancellation happens between the new monomials with 2 variables and the monomials in SP.

The size of SP_i increases during the substitution of the second and third stages, and it decreases during the substitution of the first stage. Thus, the maximum size of SP_i occurs when the substitution of atomic blocks polynomials in the second and third stages are done. We conclude that the space complexity of SCA-based verification for a structurally simple multiplier is $O(n^2)$. Due to the fact that time complexity depends on the size of SP_i, we can calculate the time complexity of verifying a structurally simple multiplier as follows:

$$\underbrace{\sum_{i=0}^{n^2-1}(|SP|+i)}_{\text{2nd and 3rd stages}} + \underbrace{\sum_{j=0}^{n^2-1}(|SP_{max}|-4j)}_{\text{1st stage}} = O(n^4), \tag{8}$$

where the first part of Eq. (8) is related to the size of SP_i during the substitution of atomic blocks polynomials in the second and third stages, and the second part is related to the size of SP_i during the substitution of AND gates polynomials in the first stage. Please note that the specification polynomial size $|SP|$ and the maximum size of intermediate polynomial after substituting atomic blocks in the second and third stages $|SP_{max}|$ are of $O(n^2)$ complexity. Thus, the time complexity of the backward rewriting process is $O(n^4)$. Based on the calculations in this section, we now come up with a theorem for the complexity of SCA-based verification:

Theorem 1: The SCA-based verification of an $n \times n$ structurally simple multiplier has $O(n^2)$ space complexity and $O(n^4)$ time complexity.

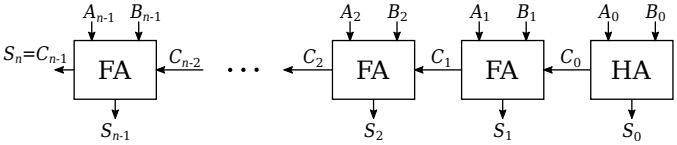

Fig. 6: n-bit carry ripple adder

VI. Polynomial Verification of FSA

The next phase of our proposed method is the verification of the original FSA. The verification method based on BDD reports very good results when it comes to the verification of integer adders. However, in order to ensure the polynomial verification, the time complexity is obtained. We first calculate the time complexity of verifying a ripple carry adder; then, we briefly review the research works on the polynomial formal verification of other adder architectures.

Fig. 6 depicts an n-bit ripple carry adder. We first calculate the computational complexity of symbolic simulation for a single FA. The sum and carry bits of a FA can be shown by ITE operations:

$$\begin{aligned} S_i &= A_i \oplus B_i \oplus C_{i-1} = ITE(C_{i-1}, A_i \odot B_i, A_i \oplus B_i) = \\ &\quad ITE(C_{i-1}, ITE(A_i, B_i, \overline{B}_i), ITE(A_i, \overline{B}_i, B_i)), \end{aligned} \tag{9}$$

$$\begin{aligned} C_i &= (A_i \wedge B_i) \vee (A_i \wedge C_{i-1}) \vee (B_i \wedge C_{i-1}) = \\ &\quad ITE(C_{i-1}, A_i \vee B_i, A_i \wedge B_i) = \\ &\quad ITE(C_{i-1}, ITE(A_i, 1, B_i), ITE(A_i, B_i, 0)) \end{aligned} \tag{10}$$

The ITE operations are computed by Algorithm 1 to get the BDDs for the S_i and C_i signals. Assuming that f, g and h are the input arguments of an ITE operator, the computational complexity is computed as $|f| \cdot |g| \cdot |h|$. As a result, the complexity of computing S_i and C_i is as follows:

$$\textit{Complexity}(S_i) = 25|C_{i-1}| + 54, \tag{11}$$

$$\textit{Complexity}(C_i) = 16|C_{i-1}| + 18, \tag{12}$$

$$\textit{Complexity}(FA_i) = 41|C_{i-1}| + 72. \tag{13}$$

It has been proved in [23] that the BDD size of the i^{th} carry bit (C_i) is bounded by $3(i+1)$. Thus, the overall complexity of verifying a ripple carry adder can be obtained as follows:

$$complexity_{[RCA]} = \sum_{i=1}^{n-1}(41|C_{i-1}| + 72) = O(n^2). \tag{14}$$

We can conclude that the order of the verification complexity is $O(n^2)$. We have also used a similar approach to prove the polynomial time complexity of verifying other adder architectures such as conditional sum adder [13] and prefix adders [14]. Since both verification complexities (i.e., SCA-based verification complexity of the structurally simple multiplier and the BDD-based verification complexity of the FSA) are polynomial, our hybrid verification approach can prove the correctness of structurally complex multipliers in polynomial space and time.

VII. Experimental Results

In this section, we evaluate the efficiency of our hybrid verification method in practice. Our method has been implemented in C++. The experiments have been carried out on an Intel(R) Xeon(R) CPU E3-1270 v3 3.50 GHz with 32 GByte of main memory.

Table I reports the results of verifying various multiplier architectures. The *Time-Out* (TO) has been set to 150 hours

TABLE I: Results of verifying different multiplier architectures

Benchmark	Size	#Gates	Run-times (seconds)							
			Proposed method	Commercial	[19]	[20]	[21]	[22]	[17]	[7]
$SP{\circ}DT{\circ}LF$	$64{\times}64$	32,680	1.72	TO	TO	TO	TO	TO	2,105.74	5.09
$SP{\circ}WT{\circ}CL$		52,083	1.84	TO	TO	TO	TO	TO	TO	16.53
$SP{\circ}BD{\circ}KS$		34,065	1.77	TO	TO	TO	TO	TO	TO	12.05
$SP{\circ}AR{\circ}CK$		31,944	1.71	TO	TO	TO	TO	TO	TO	3.07
$BP{\circ}AR{\circ}RC$		24,442	4.54	TO	37.18	TO	0.09	TO	882.52	14.36
$BP{\circ}CT{\circ}BK$		21,872	4.57	TO	TO	TO	TO	TO	1,729.33	28.53
$BP{\circ}OS{\circ}CU$		26,821	4.55	TO	TO	TO	TO	TO	TO	23.73
$BP{\circ}WT{\circ}CS$		24,830	4.59	TO	TO	TO	TO	TO	TO	41.48
$SP{\circ}WT{\circ}BK$	$128{\times}128$	131,683	16.38	TO	TO	TO	TO	TO	TO	142.22
$SP{\circ}DT{\circ}LF$		131,297	17.04	TO	TO	TO	TO	TO	TO	153.60
$SP{\circ}WT{\circ}BK$	$256{\times}256$	526,520	98.07	TO	TO	TO	TO	TO	TO	3,773.21
$SP{\circ}DT{\circ}LF$		525,531	98.58	TO	TO	TO	TO	TO	TO	5,622.45
$SP{\circ}WT{\circ}BK$	$512{\times}512$	2,103,610	827.39	TO	TO	TO	TO	TO	TO	67,493.30
$SP{\circ}DT{\circ}LF$		2,101,205	836.12	TO	TO	TO	TO	TO	TO	114,257.87

Stage 1 ⇒ **SP**: Simple partial product generator **BP**: Booth partial product generator TO: Time-Out (150 hrs)
Stage 2 ⇒ **AR**: Array **BD**: Balanced delay tree **DT**: Dadda tree **WT**: Wallace tree **CT**: Compressor tree **OS**: Overturned-stairs tree
Stage 3 ⇒ **RC**: Ripple carry **BK**: Brent-Kung **LF**: Ladner-Fischer **CL**: Carry look-ahead **KS**: Kogge-Stone **CK**: Carry-skip **CS**: Carry select **CU**: Conditional sum

for all experiments. The first column of Table I presents the architecture of the multiplier based on its stages (see abbreviations below the table). The second column *Size* shows the size of the multiplier based on the input bits. The number of gates for each architecture is given in the third column *#Gates*.

The fourth column of Table I reports the run-time of our proposed method. The remaining columns present the run-times of the most recent state-of-the-art formal verification methods. As can be seen, our approach can verify all multipliers with different architectures and sizes. It outperforms all the existing state-of-the-art formal verification methods.

The run-times of seven state-of-the-art techniques are reported in the table: While *Commercial* reports the run-times of a commercial verification tool, the remaining subcolumns give the run-times of some of the most recent SCA verification approaches. The commercial tool only verifies 16×16 multipliers. The verification methods of [19]–[22] only verify a few architectures and time-out for the rest. The proposed method in [17] supports the verification of more architectures. Finally, RevSCA 2.0 [7] verifies all multiplier architectures; however, its run-time is huge especially for the architectures bigger than 128×128 input sizes.

VIII. Conclusion

In this paper, we first proved that the SCA-based verification complexity of a structurally complex multiplier is exponential. Then, we proposed a polynomial approach to proving the correctness of structurally complex multipliers. In our method, the FSA stage is first replaced with a ripple carry adder; then, the obtained structurally complex multiplier is verified using SCA. Subsequently, the correctness of the original FSA stage is ensured using BDDs. We proved that both verification steps have polynomial complexity.

In our future research, we plan to take advantage of our hybrid verification strategy in proving the correctness of other complex designs such as floating-point arithmetic circuits. We also investigate the complexity bounds for various verification methods when it comes to proving the correctness of different classes of structures. The examples of these classes are totally symmetric functions and tree-like circuits, which have been partially studied in [24] and [25], respectively.

References

[1] R. E. Bryant, "Binary decision diagrams and beyond: enabling technologies for formal verification," in *ICCAD*, 1995, pp. 236–243.
[2] S. Disch and C. Scholl, "Combinational equivalence checking using incremental SAT solving, output ordering, and resets," in *ASP-DAC*, 2007, pp. 938–943.
[3] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT for combinational equivalence checking," in *DATE*, 2001, pp. 114–121.
[4] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers," in *ICCAD*, 2018, pp. 129:1–129:8.
[5] ——, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *DAC*, 2019, pp. 185:1–185:6.
[6] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *DATE*, 2020, pp. 544–549.
[7] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA-2.0: SCA-based formal verification of non-trivial multipliers using reverse engineering and local vanishing removal," *TCAD*, 2021.
[8] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *FMCAD*, 2019, pp. 28–36.
[9] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *TCAD*, vol. 35, no. 12, pp. 2131–2142, 2016.
[10] R. Drechsler, B. Becker, and S. Ruppertz, "The K*BMD: A verification data structure," *IEEE Design & Test of Computers*, vol. 14, no. 2, pp. 51–59, 1997.
[11] R. Drechsler, "PolyAdd: Polynomial formal verification of adder circuits," in *DDECS*, 2021, pp. 99–104.
[12] R. Drechsler, A. Mahzoon, and L. Weingarten, "Polynomial formal verification of arithmetic circuits," in *ICCIDE*, 2021, pp. 457–470.
[13] A. Mahzoon and R. Drechsler, "Late breaking results: Polynomial formal verification of fast adders," in *DAC*, 2021, pp. 1376–1377.
[14] ——, "Polynomial formal verification of prefix adders," in *ATS*, 2021, pp. 85–90.
[15] M. Keim, R. Drechsler, B. Becker, M. Martin, and P. Molitor, "Polynomial formal verification of multipliers," *Formal Methods in System Design: An International Journal*, vol. 22, no. 1, pp. 39–58, 2003.
[16] M. Barhoush, A. Mahzoon, and R. Drechsler, "Polynomial word-level verification of arithmetic circuits," in *MEMOCODE*, 2021, pp. 1–9.
[17] 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.
[18] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *DAC*, 1990, pp. 40–45.
[19] 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.
[20] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD*, 2017, pp. 23–30.
[21] C. Yu, M. Ciesielski, and A. Mishchenko, "Fast algebraic rewriting based on and-inverter graphs," *TCAD*, vol. 37, no. 9, pp. 1907–1911, 2017.
[22] D. Ritirc, A. Biere, and M. Kauers, "Improving and extending the algebraic approach for verifying gate-level multipliers," in *DATE*, 2018, pp. 1556–1561.
[23] I. Wegener, *Branching Programs and Binary Decision Diagrams*. SIAM, 2000.
[24] R. Drechsler and C. Dominik, "Edge verification: Ensuring correctness under resource constraints," in *SBCCI*, 2021.
[25] R. Drechsler, "Polynomial circuit verification using BDDs," in *ICEECCOT*, 2021, pp. 466–483.