Preserving Design Hierarchy Information for Polynomial Formal Verification

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

***Abstract*—With the growing complexity of digital circuits, formal verification has become a crucial task after the design in order to ensure the correctness of a circuit. Many existing verification methods suffer from unpredictability in their performance. It is not clear whether they have to be run for seconds, hours, or days to return the verification results or whether they fail in the end. The unpredictability can only be resolved by ensuring complexity bounds. To guarantee scalability we are in particular interested in Polynomial Formal Verification (PFV), where the space and time complexities have polynomial bounds with respect to the size of the circuit.**

The information about the design hierarchy is usually vital for PFV. Complex digital circuits consist of several components that cannot be verified with an individual verification technique in polynomial space and time. However, with additional knowledge about the boundaries of components, PFV becomes possible through the step-wise verification of sub-components and the use of different formal proof engines. In this paper, we first introduce PFV and clarify its importance. We consider the verification of several flattened gate-level arithmetic circuits and illustrate the challenges of PFV when no design hierarchy is available. Then, we show how these challenges can be overcome by preserving design hierarchy information including the boundaries of the components.

I. Introduction

In the last 30 years, the verification community has achieved many successes in proving the correctness of a wide variety of digital circuits. Several formal methods based on equivalence checking, model checking, and theorem proving have been proposed to verify both combinational and sequential circuits. Particularly, the formal verification of arithmetic circuits has gotten a lot of attention due to the high complexity and big size of these circuits: (a) *Binary Decision Diagram* (BDD) [1] and SAT-based [2], [3] verification methods report very good results for different types of adder architectures, (b) *Multiplicative Binary Moment Diagrams* (*BMDs) [4], [5] are used to verify structurally simple multipliers, and (c) *Symbolic Computer Algebra* (SCA) [6], [7], [8] is employed to verify structurally complex multipliers and dividers.

However, the main shortcoming of these techniques is unpredictability in performance, leading to several verification problems:

- It cannot be predicted before actually invoking the verification tool whether it will successfully terminate or run for an indefinite amount of time.
- The scalability of these techniques remains unknown, i.e., it is not predictable how much the run-time and the required memory increase when the size of the circuit grows.
- It is not possible to compare the performance of verification methods for a specific design and choose the best one.

In order to resolve the unpredictability of a verification method, its time and space complexities have to be calculated. Knowing the complexity bounds for a verification technique alleviates the three aforementioned verification problems. We are particularly interested in space and time complexities with the smallest possible polynomial order, i.e. $O(n^c)$, where n is a circuit parameter (e.g. the number of input bits) and c is a positive number. The concept of *Polynomial Formal Verification* (PFV) was first introduced in [9]. A formal verification method with a polynomial complexity (time and space), where the exponent in the polynomial is not too high, is scalable and can be carried out successfully for different circuit sizes.

In many cases, it is impossible to ensure the polynomial bounds for a pure gate-level circuit without any design information. Complex digital circuits usually consist of several sub-components that cannot be verified with an individual formal method in polynomial space and time. Unfortunately, the boundaries of these sub-components are lost during the synthesis to the gate-level netlist. However, we can overcome the verification challenge by preserving design hierarchy information, including boundaries of sub-components. Thus, PFV becomes possible through the step-wise verification of sub-components and the use of different formal techniques.

In this paper, we illustrate how preserving the design hierarchy information helps with PFV of complex arithmetic circuits consisting of multiplication and addition operations, e.g., $A \times B + C \times D$. We first demonstrate the challenges of verifying a complex arithmetic circuit when no design hierarchy information is available. Then, we propose a hybrid technique based on SCA and BDDs to verify complex arithmetic circuits in polynomial space and time when the design hierarchy information, including the boundaries between stages and components, is at hand. We also calculate the upper-bound space and time complexities for our case study, i.e., $A \times B + C \times D$, consisting of 2 multiplication and one addition operations. The experimental results confirm that our hybrid method can verify $A \times B + C \times D$ arithmetic circuits with up to 256-bit per input.

II. Related Works

In the last few years, researchers have come up with various PFV methods to resolve the verification unpredictability. It includes 1) proving the polynomial bounds for existing verification methods and 2) improving and extending existing formal methods to obtain polynomial upper-bound complexities.

PolyAdd [9] for the first time proved that the formal verification of three adder architectures (i.e., ripple carry adder, conditional sum adder, and carry look-ahead adder) 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 did not provide the upper-bound complexities. The authors of [10] and [11] extended PolyAdd by obtaining the upper-bound time complexities of conditional sum adder and parallel prefix adders (i.e., serial prefix adder, Ladner-Fischer adder, and Kogge-Stine adder). They calculated the time complexities by adding up the computational complexity of *If-Then-Else* (ITE) operation in each step of the symbolic simulation. Formal verification of AI-generated prefix adders in polynomial time was investigated in [12]. The authors of [13] proved that PFV of a simple *Arithmetic Logic Unit* (ALU), consisting of arithmetic and logic operations, is possible. Authors of [14] focused on PFV of approximate adders. They proved that the upper-bound time complexities of verifying approximate ripple carry adder, conditional sum adder, and carry look-ahead adder, as well as handcrafted approximate adders, are polynomial using BDDs.

The work of [15] considered PFV of a multiplier for the first time. The authors demonstrated that the verification of a Wallace-like multiplier can be carried out in polynomial space and time using *BMDs. The proof was extended by [16] to arithmetic circuits consisting of multiplication and addition operations. Moreover, the authors showed that PFV can be also performed using SCA. The authors of [17] proved that SCA-based methods have exponential upper-bound complexities when it comes to verifying structurally complex multipliers. Then, they came up with a hybrid formal method based on SCA and BDDs to achieve polynomial bounds.

In addition to arithmetic circuits, there have been some efforts to make PFV possible for other types of circuits. The authors of [18] and [19] prove that proving the correctness of totally symmetric functions and tree-like circuits is possible in polynomial space and time using BDDs. The work of [20] proposed two methods to generate polynomially verifiable circuits for an approximate function.

While there has been significant progress in PFV of various circuit types, the importance of design information has not been yet fully investigated. In this paper, we illustrate the necessity of preserving design hierarchy information in PFV of complex arithmetic circuits.

III. Background

In this section, we first explain the structure of a circuit consisting of multiplication and addition operations. Then, we review the SCA- and BDD-based verification techniques.

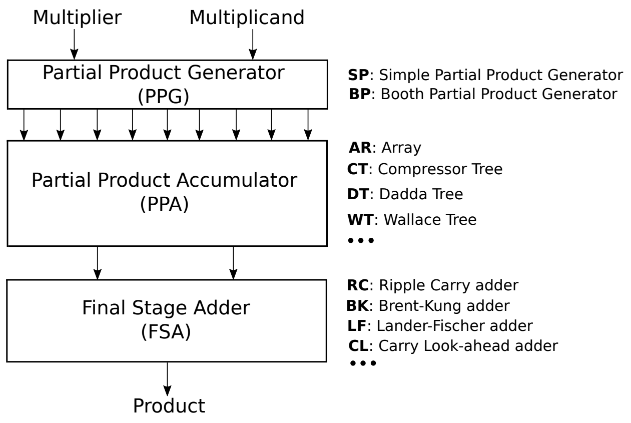

Fig. 1. General multiplier structure

A. Arithmetic Circuits 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. The architectures for the PPA stage are usually made of half-adders and full-adders. However, the only FSA architecture which consists of half-adders and full-adders is the ripple carry adder. The other existing FSA architectures have some extra logic in addition to half-adders and full-adders.

A general arithmetic circuit performs a polynomial operation on its inputs by using multiplications and additions, e.g., $Z = A \times B + C \times D$ and $Z = A^2 \times D + A \times B^2 + C$ are the examples of arithmetic circuits with A, B, C, and D as inputs and Z as output. There are two possible ways to generate arithmetic circuits:

- The partial products for all multiplication operations are generated; then, they are reduced to only two rows of partial products using multi-operand adders in the PPA stage. Finally, the two rows of partial products are added up using the FSA stage to generate the final product. Fig. 2 shows the structure of an arithmetic circuit that performs $Z = A \times B + C \times D$ operation.
- The multiplication and addition operations are implemented individually; then, they are connected in a way to correctly implement the circuit function. Fig. 3 depicts the structure of an arithmetic circuit that performs $Z = A \times B + C \times D$ operation by combining individual multiplication and addition blocks.

In this paper, we use $Z = A \times B + C \times D$ as our case study and investigate its PFV. Nevertheless, the proposed techniques can be easily used for other arithmetic circuits.

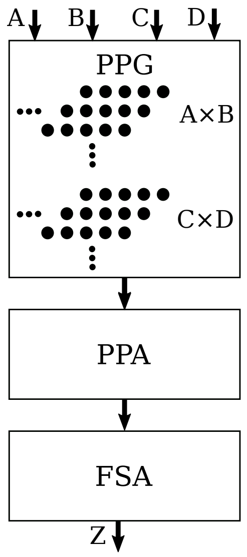

Fig. 2. $Z = A \times B + C \times D$ implementation in three stages

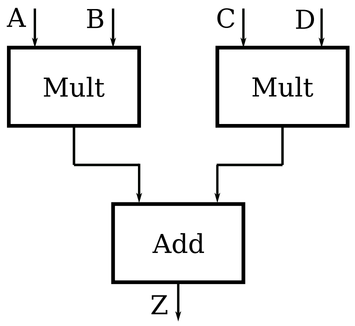

Fig. 3. $Z = A \times B + C \times D$ implementation using individual blocks

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. 4 $SP = 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (2A_1 + A_0)(2B_1 + B_0)$, 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:

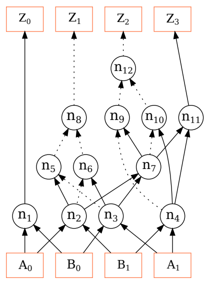

Fig. 4. 2×2 mult

$$
\begin{aligned}
& SP := 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - A \times B \\
& SP \xrightarrow{P_{Z_3}} SP_1 := 8n_{11} + 4Z_2 + 2Z_1 + Z_0 - A \times B \\
& SP_1 \xrightarrow{P_{Z_2}} SP_2 := 8n_{11} + 4 - 4n_{12} + 2Z_1 + Z_0 - A \times B \\
& SP_2 \xrightarrow{P_{n_{12}}} SP_3 := 8n_{11} + 4n_9 + 4n_{10} - 4n_9 n_{10} + 2Z_1 + Z_0 - A \times B \\
& SP_3 \xrightarrow{P_{n_{11}}} SP_4 := 8n_4 n_7 + 4n_9 + 4n_{10} - 4n_9 n_{10} + 2Z_1 + Z_0 - A \times B \\
& \quad \vdots \\
& SP_{13} \xrightarrow{P_{n_3}} SP_{14} := n_1 + n_2 - (A_0 B_0 + A_0 B_1) \\
& SP_{14} \xrightarrow{P_{n_2}} SP_{15} := n_1 - (A_0 B_0) \\
& SP_{15} \xrightarrow{P_{n_1}} r := 0
\end{aligned}
$$

Fig. 5. Backward rewriting steps

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

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 [21], [22]).

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 [7] for more details).

The step-wise division of SP by node polynomials is shown in Fig. 5 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. 5 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.

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 $R$
```

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:

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

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

$$\begin{aligned} f \wedge g &= ITE(f, g, 0), & f \vee g &= ITE(f, 1, g), \\ f \oplus g &= ITE(f, \overline{g}, g), & \bar{f} &= ITE(f, 0, 1). \end{aligned} \tag{3}$$

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 [23].

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.

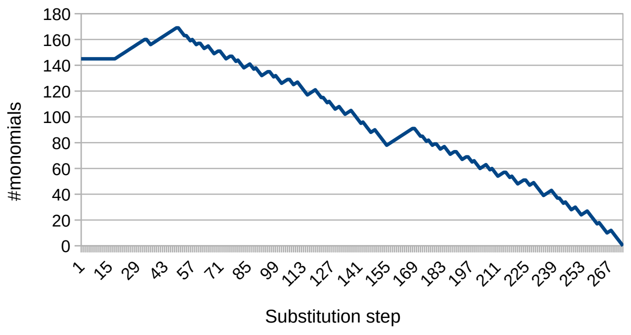

Fig. 6. Number of monomials at each step of substitution for (1) architecture

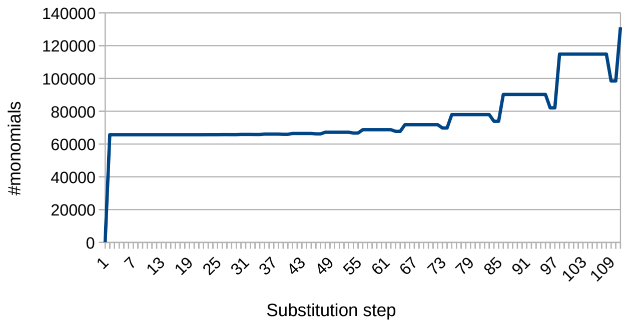

Fig. 7. Number of monomials at each step of substitution for (2) architecture

IV. Verification Challenges

It has been shown in [17] that the SCA-based verification method reports very good results for the multipliers whose second and third stages are only made of half-adders and full-adders. However, if there are some extra gates/nodes in addition to half-adders and full-adders, an explosion happens in the size of intermediate polynomials during backward rewriting. The same behavior can be also observed for the verification of general arithmetic circuits.

We provide the experimental evidence for the verification of $Z = A \times B + C \times D$ with 8×8 input size when two different architectures are used to implement the function. These architectures are:

1) The implementation is done based on Fig. 3. The architecture for the 8-bit multipliers is *simple partial product generator* ∘ *dadda* ∘ *ripple carry adder* ($SP{\circ}DT{\circ}RC$). The 16-bit adder has *ripple carry adder* architecture.
2) The implementation is done based on Fig. 3. The architecture for the 8-bit multipliers is *simple partial product generator* ∘ *dadda* ∘ *carry look-ahead adder*

($SP \circ DT \circ CL$). The 16-bit adder has *carry look-ahead adder* architecture.

Fig. 6 and Fig. 7 depicts the size of the intermediate polynomial at each step of backward rewriting for the (1) and (2) architectures, respectively. There is a slight increase in the number of monomials during the verification of the (1) architecture in Fig. 6, and then it decreases until the zero polynomial is obtained. Thus, the SCA-based verification is carried out successfully. On the other hand, there is an explanation in the number of monomials during the verification of the (2) architecture in Fig. 7. As a result, the verification process cannot be completed. This explosion can be observed during the verification of arithmetic circuits whose adder architectures contain extra gates/nodes in addition to half-adders and full-adders, e.g., carry look-ahead adder. In the next section, we show that if the design hierarchy information is available, we can overcome the verification challenge by introducing a hybrid verification technique based on SCA and BDDs.

V. PFV with Design Hierarchy Information

In this section, we first introduce our hybrid technique based on SCA and BDDs to verify general arithmetic circuits when the design hierarchy information is available. Then, we prove that the proposed technique has polynomial upper-bound complexities.

A. Methodology

It has been shown in [6], [17] that the adder circuits are the main challenge during the SCA-based verification of complex multipliers. The only adder which can be successfully handled by the SCA-based method is the ripple carry adder since it is only made of half-adders and full-adders. It is also the case for the verification of general arithmetic circuits. If only ripple carry adders are used in the circuit, it can be verified using the SCA-based method (see Fig. 6). Otherwise, an explosion happens during backward rewriting (see Fig. 7). We overcome this challenge by introducing a hybrid verification technique in the presence of design hierarchy information. Our proposed method consists of three steps:

1) replacing adder circuits with ripple carry adders,
2) verifying original adders using BDDs, and
3) verifying the new arithmetic circuit using SCA.

If the arithmetic circuit is implemented using the three-stage structure of Fig. 2, the only adder circuit is the FSA. Since the design hierarchy information is available, we can simply replace it with a ripple carry adder. On the other hand, it is possible that the arithmetic circuit is implemented by individual blocks as shown in Fig. 3. Thus, we replace the FSA stage of each multiplier block with a ripple carry adder. In addition, we also replace the individual adder blocks with ripple carry adders.

We now prove that for our case study, i.e., $Z = A \times B + C \times D$ arithmetic circuit, the whole verification is polynomially bounded when the design hierarchy information is available. The proof can be then easily extended to other general arithmetic circuits.

B. Polynomial BDD-based Verification of Arithmetic Circuit

After replacing adders with ripple carry adders, the complex arithmetic circuit is now converted into a structurally simple circuit since, except for the PPG architecture, the rest of the circuit is only made of half-adders and full-adders. We now calculate the space and time complexity of SCA-based verification when it comes to our case study, i.e., $Z = A \times B + C \times D$. We focus on the implementation in Fig. 3 while it can be also extended to Fig. 2 implementation.

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. During backward rewriting of structurally simple arithmetic circuits, 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.

The second and third stages of the two multipliers and the adder in Fig. 3 are only made of atomic blocks (i.e., half-adders and full-adders). 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 and a full-adder are as follows:

$$\begin{aligned} HA(in: X, Y \quad out: C, S) \quad &\Rightarrow \quad 2C + S = X + Y \\ FA(in: X, Y, Z \quad out: C, S) \quad &\Rightarrow \quad 2C + S = X + Y + Z. \end{aligned} \tag{5}$$

Eq. (5) 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 $k \leqslant 1$. For the arithmetic circuit in Fig. 3, the number of atomic blocks in the Add and Mult blocks are n and n^2, respectively. Thus, the total number of atomic blocks is $O(n^2)$. Similarly, the size of the specification polynomial (i.e. $SP := Z - A \times B - C \times D$) is $O(n^2)$. After substitution of atomic blocks in the ADD and Mult blocks, the size of SP_i increases by $O(n^2)$.

The first stages of two Mult blocks contain $2 \times n^2$ 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. Due to the fact that time complexity depends on the size of SP_i, we can calculate the time complexity of verifying the $Z = A \times B + C \times D$ arithmetic circuit in Fig. 3 as follows:

$$\underbrace{\sum_{i=0}^{2n^2+n-1} (|SP| + i)}_{\text{atomic blocks}} + \underbrace{\sum_{j=0}^{2n^2-1} (|SP_{max}| - 4j)}_{\text{1st stage of multipliers}} = O(n^4). \quad (6)$$

The first part of Eq. (6) is related to the size of SP_i during the substitution of atomic blocks polynomials in ADD and Mult blocks. Thus, each substitution step increases the size of SP_i by a maximum of one variable. The second part of Eq. (6) is related to the size of SP_i during the substitution of AND gates polynomials in the first stages of Mult blocks. Please note that the specification polynomial size $|SP|$ and the maximum size of intermediate polynomial after substituting atomic blocks $|SP_{max}|$ has $O(n^2)$ complexity. Thus, the time complexity of the backward rewriting process is $O(n^4)$. Thus, PFV of the arithmetic circuit is possible after the adders substitution.

C. *Polynomial BDD-based Verification of Adders*

The next phase of our method is the verification of the original adders. The verification method based on BDD reports very good results when it comes to the verification of integer adders. However, in order to ensure PFV, the time complexity has to be obtained. The polynomial complexity of symbolic simulation has been proven for three adders in [9]. The authors of [10] and [11] calculated the time complexity of symbolic simulation for conditional sum adders and parallel prefix adders, respectively. It has been shown in these research works that the upper bound complexity does not exceed $O(n^4)$. Thus, PFV of adder architectures is possible using BDDs. The calculations are done based on the computational complexity of ITE operation at each step of the symbolic simulation.

VI. Experimental Results

In this section, we evaluate the efficiency of our verification method when the design hierarchy information is available. 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. We have generated the $Z = A \times B + C \times D$ arithmetic circuits with various adder architectures using a modified version of GenMul [24].

TABLE I
Result of verifying $Z = A \times B + C \times D$ arithmetic circuit

Benchmark	Size	#Gates	**Run-times (seconds)**	
			Ours	Commercial
Ladner-Fischer	16	7,192	0.07	74.00
Carry look-ahead		7,705	0.18	78.00
Kogge-Stone		7,509	0.11	70.00
Carry-skip		7,332	0.11	81.00
Ripple carry		6,821	0.14	75.00
Brent-Kung		7,155	0.09	70.00
Conditional sum		7,369	0.13	79.00
Carry select		7,212	0.11	74.00
Ladner-Fischer	32	25,688	0.44	TO
Carry look-ahead		30,487	0.68	TO
Kogge-Stone		28,570	0.52	TO
Carry-skip		26,978	0.49	TO
Ripple carry		22,921	0.58	TO
Brent-Kung		25,120	0.42	TO
Conditional sum		27,044	0.47	TO
Carry select		25,701	0.45	TO
Ladner-Fischer	64	97,504	3.41	TO
Carry look-ahead		107,447	4.45	TO
Kogge-Stone		102,178	3.51	TO
Carry-skip		99,008	3.08	TO
Ripple carry		91,519	3.74	TO
Brent-Kung		97,274	4.03	TO
Conditional sum		99,486	3.29	TO
Carry select		98,236	3.44	TO
Ladner-Fischer	128	399,760	34.04	TO
Carry look-ahead		421,068	38.21	TO
Kogge-Stone		412,588	34.88	TO
Carry-skip		405,785	33.92	TO
Ripple carry		388,817	35.11	TO
Brent-Kung		399,114	32.78	TO
Conditional sum		402,576	32.85	TO
Carry select		400,567	34.50	TO

Table I reports the results of verifying the arithmetic circuits. The *Time-Out* (TO) has been set to 24 hours for all experiments. The first column of Table I presents the adder architecture used in the arithmetic circuit. The second column *Size* shows the size of the arithmetic circuit based on the bits per input. The third column *#Gates* gives number of gates. The fourth column of Table I reports the run-times of our method and a commercial verification tool. As can be seen, our approach verifies the arithmetic circuits with various adder architectures and different sizes while the commercial tool only proves the correctness of 16-bit arithmetic circuits and times-out for the other benchmarks.

The experimental evaluations confirm that our proposed method can verify complex arithmetic circuits efficiently if the design hierarchy information is available. The PFV method can be also extended to support the verification of arithmetic circuits with arbitrary functions.

VII. Conclusion

In this paper, we illustrated the importance of design hierarchy information in PFV. The complex designs usually consist of many sub-components, which can be verified in polynomial space and time using a suitable verification technique. However, accessing these components is not always possible after synthesizing the circuit into a gate-level netlist. This problem can be alleviated by preserving the design hierarchy information. We proposed a hybrid verification method to verify general complex arithmetic circuits in polynomial space and time when the design hierarchy information is available.

In future research, we plan to investigate the PFV of other digital circuits (i.e., complex processors) when the design hierarchy information is available.

Acknowledgement

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

References

[1] S. Malik, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Logic verification using binary decision diagrams in a logic synthesis environment," in *International Conference on Computer-Aided Design*, 1988, pp. 6–9.

[2] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Design Automation Conference*, 1997, pp. 263–268.

[3] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Design Automation Conference*, 2006, pp. 532–535.

[4] K. Hamaguchi, A. Morita, and S. Yajima, "Efficient construction of binary moment diagrams for verifying arithmetic circuits," in *International Conference on Computer-Aided Design*, 1995, pp. 78–82.

[5] R. E. Bryant and Y. A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *Design Automation Conference*, 1995, pp. 535–541.

[6] 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," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, pp. 1573–1586, 2022.

[7] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *Formal Methods in Computer-Aided Design*, 2019, pp. 28–36.

[14] M. Schnieber, S. Fröhlich, and R. Drechsler, "Polynomial formal verification of approximate adders," in *EUROMICRO Symposium on Digital System Design*, 2022.

[8] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, vol. 35, no. 12, pp. 2131–2142, 2016.

[9] R. Drechsler, "PolyAdd: Polynomial formal verification of adder circuits," in *IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2021, pp. 99–104.

[10] A. Mahzoon and R. Drechsler, "Late breaking results: Polynomial formal verification of fast adders," in *Design Automation Conference*, 2021, pp. 1376–1377.

[11] A. Mahzoon and R. Drechsler, "Polynomial formal verification of prefix adders," in *Asian Test Symp.*, 2021, pp. 85–90.

[12] R. Drechsler and A. Mahzoon, "Towards polynomial formal verification of AI-generated arithmetic circuits," in *International Symposium on Devices, Circuits and Systems*, 2021.

[13] R. Drechsler, A. Mahzoon, and L. Weingarten, "Polynomial formal verification of arithmetic circuits," in *International Conference on Computational Intelligence and Data Engineering*, 2021, pp. 457–470.

[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 *ACM & IEEE International Conference on Formal Methods and Models for Codesign*, 2021, pp. 1–9.

[17] R. Drechsler, A. Mahzoon, and M. Goli, "Towards polynomial formal verification of complex arithmetic circuits," in *IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2022, pp. 1–6.

[18] R. Drechsler and C. Dominik, "Edge verification: Ensuring correctness under resource constraints," in *Symposium on Integrated Circuits and System Design*, 2021, pp. 1–6.

[19] R. Drechsler, "Polynomial circuit verification using BDDs," in *International Conference on Electrical, Electronics, Communication, Computer Technologies and Optimization Techniques*, 2021, pp. 466–483.

[20] M. Schnieber, S. Fröhlich, and R. Drechsler, "Polynomial formal verification of approximate functions," in *IEEE Annual Symposium on VLSI*, 2022.

[21] 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 *Design, Automation and Test in Europe*, 2016, pp. 1048–1053.

[22] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers," in *International Conference on Computer-Aided Design*, 2018, pp. 129:1–129:8.

[23] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Design Automation Conference*, 1990, pp. 40–45.

[24] A. Mahzoon, D. Große, and R. Drechsler, "GenMul: Generating architecturally complex multipliers to challenge formal verification tools," in *Recent Findings in Boolean Techniques*, R. Drechsler and D. Große, Eds. Springer, 2021, pp. 177–191.