Towards Formal Verification of Optimized and Industrial Multipliers

Alireza Mahzoon[1] Daniel Große[1,2] Christoph Scholl[3] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
[3]Institute of Computer Science, University of Freiburg, Germany
{mahzoon,grosse,drechsle}@informatik.uni-bremen.de, scholl@informatik.uni-freiburg.de

***Abstract*—Formal verification methods have made huge progress over the last decades. However, proving the correctness of arithmetic circuits involving integer multipliers still drives the verification techniques to their limits. Recently, *Symbolic Computer Algebra* (SCA) methods have shown good results in the verification of both large and non-trivial multipliers. Their success is mainly based on (1) reverse engineering and identifying basic building blocks, (2) finding converging gate cones which start from the basic building blocks and (3) early removal of redundant terms (vanishing monomials) to avoid the blow-up during backward rewriting.**

Despite these important accomplishments, verifying optimized and technology-mapped multipliers is an almost unexplored area. This creates major barriers for industrial use as most of the designs are area and delay optimized. To overcome the barriers, we propose a novel SCA-method which supports the formal verification of a large variety of optimized multipliers. Our method takes advantage of a dynamic substitution ordering to avoid the monomial explosion during backward rewriting. Experimental results confirm the efficiency of our approach in the verification of a wide range of optimized multipliers including industrial benchmarks.

I. Introduction

An integer multiplier is one of the most frequent units in many applications (e.g. signal processing, cryptography, and machine learning). The wide variety of multiplication algorithms and a large number of building blocks make it one of the most complex parts of many designs. Integer multipliers are usually optimized in terms of area and delay particularly in industrial applications where having an efficient design is critical. Formal verification of the optimized multipliers is highly important to ensure the correctness of the circuit after optimization. However, the effects of optimization on the multiplier structure make the verification a tough challenge.

In the last 20 years, several formal verification methods have been proposed to prove the correctness of integer multipliers. However, they have serious disadvantages: (a) *Decision Diagrams* (DDs) (such as BDDs and *BMDs) [1] suffer from memory blow-up when the multipliers are large, (b) *Boolean satisfiability* (SAT) and *Satisfiability Modulo Theories* (SMT) face exponential run-times when the bit-width increases, (c) *Theorem Proving* [2] needs manual effort, and (d) *Term Rewriting* [3] is not fully automated as a manual update of rewrite rules is necessary for implementations which are not yet represented in the database.

Recently, *Symbolic Computer Algebra* (SCA) approaches have addressed many challenges of formal verification, see for instance [4], [5], [6], [7], [8], [9], [10], [11], [12]. SCA-based multiplier verification generally consists of three main steps: 1) representing the function of the multiplier based on its inputs and outputs as a *Specification Polynomial* SP, 2) capturing the logical gates (nodes) as a set of polynomials P_G, and 3) proving the membership of SP in the ideal generated by P_G using Gröbner basis theory.

In the just mentioned 3rd step, SP is divided by P_G in a step-wise process called *backward rewriting*, and eventually, the remainder is evaluated. If this remainder is zero, the multiplier is correct; otherwise, it is buggy.

The local removal of redundant monomials (so-called *vanishing monomials*) and reverse engineering have boosted the efficiency of SCA-based methods in verifying both large and non-trivial multipliers. Recently, [10] showed that (i) the vanishing monomials cause an explosion in the number of monomials during backward rewriting of non-trivial multipliers, and (ii) the vanishing monomials are formed when substituting a converging gate, i.e. a gate to which both outputs of a *Half Adder* (HA) converge. Hence, algorithms for detecting *Converging Gate Cones* (CGCs) and early removal of vanishing monomials have been proposed to make the global backward rewriting phase vanishing-free. Furthermore, [13] demonstrated that identifying basic building blocks called atomic blocks, e.g. HAs, *Full Adders* (FAs), etc., is essential in revealing all vanishing monomials and finally to speed up the overall verification process.

Despite the progress in formal verification of multipliers, still proving the correctness of optimized designs is an almost unexplored area. Applying the state-of-the-art SCA-based verification methods to optimized multipliers leads to an explosion in the backward rewriting phase. The reason for the explosion is that optimization typically destroys the boundaries of several atomic blocks. As a consequence, the compact word-level polynomials for the "lost atomic blocks" are no longer available during backward rewriting. Likewise the static substitution order for backward rewriting of the state-of-the-art SCA-verifiers becomes now susceptible to the ordering of the nodes for lost atomic blocks.

Contribution: In this paper, we introduce the novel SCA-method DyPoSub[1] to verify optimized and industrial multipliers. We first investigate how optimization changes the structure of a multiplier netlist. As a result of these changes, the state-of-the-art SCA-verifiers fail and we analyze the reasons, i.e. in particular the effect of destroyed boundaries for atomic blocks during backward rewriting. We then show that this effect can be mitigated by a **dynamic substitution order** which

This work was supported by the University of Bremen's graduate school SyDe funded by the German Excellence Initiative, and by the German Academic Exchange Service (DAAD).

[1]Our tool DyPoSub is available on GitHub; links can be found at http://www.sca-verification.org/dyposub

978-3-9819263-4-7/DATE20/©2020 EDAA

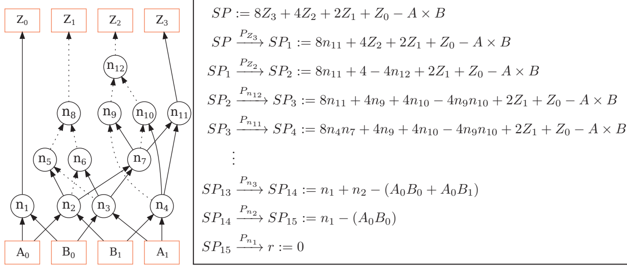


Fig. 1. 2×2 mult

Fig. 2. Backward rewriting steps

allows keeping the size of the current specification polynomial moderate. To the best of our knowledge, a dynamic substitution order has not been considered before in SCA-verification. The experimental results confirm that our proposed method can verify a wide range of optimized multipliers including industrial benchmarks which was not possible before.

Related Work: As mentioned above, a series of SCA-based verification approaches emerged for integer multipliers in the last five years [4], [5], [6], [7], [8], [10], [11], [12]. While there has been significant progress from simple multiplier architectures to complex architectures as well as in the size of the multipliers, multipliers processed by strong logic synthesis and technology mapping have been still out of reach for the existing approaches. Moreover, the idea of dynamic substitution ordering has not been investigated so far.

II. Preliminaries

A. Multiplier Structure

Most integer multipliers consist of three stages: (1) *Partial Product Generator* (PPG) which creates partial products from two inputs, (2) *Partial Product Accumulator* (PPA) which reduces partial products and computes their sums, and (3) *Final Stage Adder* (FSA) which converts these sums to the corresponding binary output.

We use an *AND-Inverter Graph* (AIG) representation of a multiplier as input for our verification method (Fig. 1 shows the AIG of a 2×2 multiplier). A great advantage of AIGs is the possibility of advanced reverse engineering. Based on cut enumeration, atomic blocks can be identified very fast, even in multipliers with millions of nodes [14], [13].

B. SCA Verification

First, we briefly summarize some basics:

- **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. $P = c_1M_1 + \dots + c_jM_j$ 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 AIG evaluate the *Specification Polynomial* (SP) to 0. The SP determines the function of a multiplier based on its inputs and outputs, e.g. for the 2×2 multiplier of Fig. 1 $SP = 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (2A_1 + A_0)(2B_1 + B_0)$ 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 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.

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 [15], [8], [16] for more details).

The step-wise division of SP by node polynomials is shown in Fig. 2 for the 2×2 multiplier. As the remainder is zero, the circuit is bug-free. In integer multipliers, 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. 2 is equivalent to substituting n_{11} with $tail(P_{n_{11}}) = n_4n_7$ 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*. The existing SCA-based verification methods use a static reverse topological order to substitute node polynomials in the intermediate specification polynomial. We refer to this intermediate polynomial as SP_i in the rest of the paper.

During the backward rewriting of complex multipliers, a huge number of redundant monomials known as vanishing monomials are generated. Removing the vanishing monomials is essential to avoid an explosion in the size of SP_i. In [10] and [13] a method has been presented which first performs a local backward rewriting to remove the vanishing monomials early in the so-called *Converging Gate Cones* (CGCs). Overall, this allows a vanishing-free global backward rewriting.

III. Challenges of Verifying Optimized Multipliers

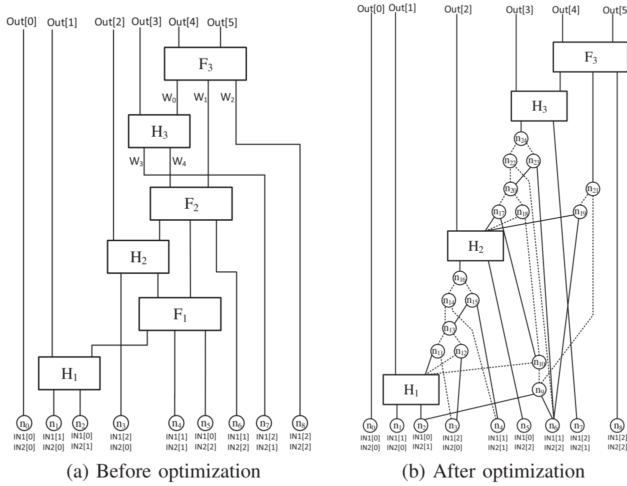
In this section, we first demonstrate how optimization changes the structure of a multiplier netlist. As a result of these changes, a major challenge arises for formal verification of multipliers which we explain next.

A. Optimization and Multiplier Structure

Multiplication algorithms determine how the building blocks are connected in each stage of a multiplier to reach a certain design goal, e.g. minimum area or minimum delay [17]. Although the type of building blocks and the connections largely vary from one algorithm to another, there are standard building blocks (also referred as atomic blocks). Examples include HAs, FAs, MUXs, and carry predictors [18].

Example 1. *Fig. 3a shows the AIG of a 3×3 array multiplier after reverse engineering and identifying the atomic blocks. The first stage of the multiplier consists of nine AND gates (i.e. $n_0, n_1, \dots, n_8$) to generate partial products. The second and third stages are made of HA and FA networks[2] to reduce partial products and perform the final addition. As can be seen*

[2]A HA is denoted as H_i and an FA as F_k, respectively.



(a) Before optimization (b) After optimization

Fig. 3. AIGs of 3×3 array multiplier

in Fig. 3a, the boundaries of atomic blocks and the connections between them are fully visible.

However, performing optimization on the multiplier netlist completely changes the situation. More precisely, the effect of optimization appears at two hierarchy levels:

1) Logic optimization inside building blocks: the number of nodes in a building block shrinks, e.g. 11 AIG nodes of an FA are reduced to 8 nodes. In this case, the overall structure of the multiplier does not change significantly (as long as the reverse engineering is still able to identify the respective atomic block).
2) Logic optimization across building blocks: blocks are merged to reduce the number of nodes and as a consequence some block boundaries are removed. In this case, identifying building blocks is no longer possible.

We now come back to the array multiplier example.

Example 2. *Fig. 3b shows the AIG for the* 3×3 *array multiplier after applying* resyn3 *optimization from abc [19]. While the overall number of AIG nodes is reduced by* 15%*, the optimization destroys the boundaries of the two FAs* F_1 *and* F_2*, respectively.*

As becomes evident from the example, there are some nodes in the circuit which have been part of atomic blocks (FAs in the example) before optimization, but they cannot be identified as any atomic blocks anymore. Moreover, in non-trivial multipliers optimization may also change the boundaries of larger blocks, e.g. n-bit MUXs, carry operators, partial product generator units. Overall, losing block boundaries poses a major challenge to SCA verification of integer multipliers which we detail in the following section.

B. Challenge for Backward Rewriting of Optim. Multipliers

An important property of atomic blocks is the compactness of the polynomials describing their input/output relations. For example, the two most common atomic blocks, i.e. HA and FA, can be expressed using the following word-level relations:

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

Let's have a look on backward rewriting when reaching atomic blocks and such word-level polynomials: Substituting an atomic block polynomial in SP_i only slightly changes the size of SP_i (i.e. number of monomials). As a result, the size of SP_i remains almost identical.

Example 3. *In Fig. 3a, substituting the HA and FA polynomials increases the size of* SP_i *by zero or one, respectively. For example, substituting the* F_3 *polynomial* ($2Out[5] + Out[4] = W_0 + W_1 + W_2$) *and the* H_3 *polynomial* ($2W_0 + Out[3] = W_3 + W_4$) *in the first and the second steps of backward rewriting results in:*[3]

$$
\begin{aligned}
&SP := 32Out[5] + 16Out[4] + 8Out[3] + \ldots \\
&SP \xrightarrow{F_3} SP_1 := 16W_2 + 16W_1 + 16W_0 + 8Out[3] + \ldots \\
&SP_1 \xrightarrow{H_3} SP_2 := 16W_2 + 16W_1 + 8W_3 + 8W_4 + \ldots
\end{aligned}
\tag{3}
$$

On the other hand, substituting later $n_0, n_1, \ldots, n_8$ *(AND gates) polynomials results in cancellation of two terms per AND gate and reduces the size of* SP_i *by two per AND gate.*

If we now look on the overall backward rewriting algorithm (and not only on a single substitution step), all existing SCA-verifiers use a static pre-determined substitution order derived from the topological sorting of the AIG nodes/circuit elements. This worked out fine so far, as sharp increases in the size of SP_i are not occurring if most of the time atomic blocks are handled as a whole – either by substituting word-level polynomials (if possible) or at least by using a substitution order that substitutes polynomials for the single outputs of atomic blocks strictly one after the other. If we lose the block boundaries in optimized multipliers, the exploitation of this knowledge is not possible anymore. Thus, we observe much larger polynomials, and in the worst-case a blow-up.

Example 4. *In Fig. 3b, after substituting the* F_3 *and* H_3 *polynomials, there are several degrees of freedom for the subsequent substitutions respecting the reverse topological order of the* $n_0, n_1, \ldots, n_{24}$ *nodes and* H_1*,* H_2 *components. A straightforward topological order results in a sudden increase in the* SP_i *size, and even an explosion in larger multipliers. For example, using a topological order* SO_1 *during the backward rewriting of an optimized* 8×8 *array multiplier results in an intermediate polynomial with* $106,938$ *monomials. However, by using a different order* SO_2 *we can limit the maximum size of* SP_i *to* 203*.*

IV. SCA VERIFICATION USING DYNAMIC BACKWARD REWRITING

In this section, we first give an overview of our proposed approach DYPOSUB. Then, we introduce dynamic backward rewriting which allows verification of optimized multipliers.

A. Overview

Algorithm 1 shows the top-level pseudo-code of our approach. The input is the AIG representation of a multiplier, and the algorithm returns *true* (*false*) if the multiplier is correct (buggy). In the first step, the specification polynomial SP is created based on the size of the multiplier (Line 1). Then, the atomic blocks are identified by reverse engineering (Line 2). Subsequently, the converging gate cones are detected in the remaining nodes which are not part of any atomic blocks

[3]The case that the left hand sides of word-level relations for atomic blocks do not occur in the needed form in the SP_i is discussed later in Section IV-B.

Algorithm 1 SCA Verification

Input: Multiplier M
Output: $TRUE$ if M is correct, and $FALSE$ otherwise
1: $SP \leftarrow$ create_specification_polynomial (M)
2: $AB \leftarrow$ reverse_engineering (M)
3: $CGC \leftarrow$ find_converging_gate_cones (M, AB)
4: $FFC \leftarrow$ find_fanout-free_cones (M, AB, CGC)
5: $C \leftarrow CGC \cup FFC$
6: $P \leftarrow$ extract_polynomials (C)
7: $P \leftarrow$ remove_vanishing_monomials (P)
8: $r \leftarrow$ dynamic_backward_rewriting (SP, AB, C, P)
9: **if** r is equal to zero **then return** TRUE **else return** $FALSE$

(Line 3). The rest of the nodes are grouped into fanout-free cones (Line 4). The polynomials for all cones are extracted (Line 5 – Line 6) and the vanishing monomials are removed (Line 7). Finally, we perform the proposed dynamic backward rewriting (for details see next section) which is based on the dynamic ordering of cone and block polynomials to obtain the remainder (Line 8). If the remainder is zero the circuit is correct; otherwise, it is buggy (Line 9).

B. Dynamic Backward Rewriting

Before introducing the dynamic backward rewriting, we first make several definitions:

Definition 1. *Atomic blocks, Converging Gate Cones (CGCs), and Fanout-Free Cones (FFCs) are called* Components. *A component has one output, if it is a CGC or an FFC; and may have several outputs if it is an atomic block. A multiplier consists of several components.*

Definition 2. *A component has a polynomial describing the (word-level) relation of output(s) and inputs. In components with one output (i.e. CGCs and FFCs), we have:*

$$Out = F(IN_1, IN_2, \ldots, IN_n) \tag{4}$$

where Out is the sole output of the component, and $F(IN_1, IN_2, \ldots, IN_n)$ is a polynomial based on the component inputs. In components with more than one output (i.e. atomic blocks), we have:

$$\begin{aligned} Out_1 &= F_1(IN_1, IN_2, \ldots, IN_n) \\ Out_2 &= F_2(IN_1, IN_2, \ldots, IN_n) \\ &\ldots \\ Out_m &= F_m(IN_1, IN_2, \ldots, IN_n) \end{aligned} \tag{5}$$

where each output can be described as a polynomial based on the primary inputs. There is also a more compact relation between outputs and inputs in these components:

$$G(Out_1, Out_2, \ldots, Out_m) = F(IN_1, IN_2, \ldots, IN_n) \tag{6}$$

where G and F are polynomials based on the outputs and inputs, respectively.

Example 5. *In an FA with the inputs X, Y, and Z, and outputs C and S, we have:*

$$C = XY + XZ + YZ - 2XYZ \tag{7}$$
$$S = X + Y + Z - 2XY - 2XZ - 2YZ + 4XYZ \tag{8}$$
$$2C + S = X + Y + Z \tag{9}$$

where (7) and (8) show the polynomials for carry and sum outputs, respectively. However, (9) indicates the compact relation between outputs and inputs.

Definition 3. *Substituting a component polynomial in the intermediate specification polynomial (i.e. SP_i) means finding and replacing all the occurrences of component output(s) by the corresponding polynomial(s) in SP_i.*

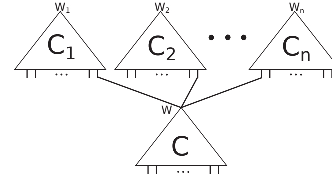


Fig. 4. Substitution candidate

Algorithm 2 Dynamic Backward Rewriting

Input: Specification Poly SP, Set of Components B, Set of Component Polys P
Output: Remainder r
1: $SP_i \leftarrow SP$
2: **while** B is not empty **do**
3: $CB \leftarrow$ Find eligible candidates in B for substitution
4: **for each** $b \in CB$ **do**
5: $O[b] \leftarrow$count_occurrence(b, SP_i)
6: $sortedCB \leftarrow$ Sort CB in ascending order based on O
7: $SP_{old} \leftarrow SP_i$
8: $threshold \leftarrow 0.1$; $j \leftarrow 0$; $no_candidate \leftarrow TRUE$
9: **while** no_candidate is TRUE **do**
10: $SP_i \leftarrow$ Substitute ($P_{sortedCB[j]}$, SP_i)
11: **if** $(size(SP_i) - size(SP_{old}))/size(SP_{old}) < threshold$ **then**
12: Remove $sortedCB[j]$ from B
13: $no_candidate \leftarrow FALSE$
14: **else**
15: $SP_i \leftarrow SP_{old}$; $j \leftarrow j + 1$
16: **if** $j \geq size(SortedCB)$ **then**
17: $j \leftarrow 0$; $threshold \leftarrow threshold \times 2$
18: $r \leftarrow SP_i$
19: **return** r

To ensure correct substitution, we need two rules: (1) In components with more than one output (i.e. atomic blocks), which have a compact word-level relation between inputs and outputs (see Equation (6)), we first search for $G(Out_1, Out_2, \ldots, Out_m)$ in SP_i. Then, we have to distinguish between two cases: If we have found $G(Out_1, Out_2, \ldots, Out_m)$, we directly substitute it by $F(IN_1, IN_2, \ldots, IN_n)$; otherwise, we substitute each output with the corresponding polynomials. Finding the exact G polynomial is sometimes not possible, particularly in optimized multipliers. Therefore, following this rule guarantees the correct substitution. (2) Assume that C is a component with output W, and $C_1, C_2, \ldots, C_n$ are components having W as one of their inputs (see Fig. 4). The C polynomial should be substituted only after substituting the $C_1, C_2, \ldots, C_n$ polynomials. Following this rule guarantees that a component polynomial needs to be substituted only once during the backward rewriting as for example the signal W never appears again in the SP_i after substituting the C polynomial. A component which satisfies this rule and is ready for substitution is called a *Candidate*.

At each step of backward rewriting, there are several substitution candidates. As we discussed in Section III, choosing a substitution order which keeps the size of SP_i as small as possible and avoids explosion in optimized multipliers is vital. Therefore, we propose dynamic backward rewriting.

Algorithm 2 shows our proposed dynamic backward rewriting. The algorithm receives the specification polynomial, the set of components, and their corresponding polynomials as inputs, and performs all possible backward rewriting steps. First, the substitution candidates are found (see Line 3 in Algorithm 2). Then, the number of times which the output of each candidate occurs in SP_i is counted (Line 4 – Line 5). We sort the candidates in ascending order based on the number of their output occurrences in SP_i (Line 6) and basically we try to substitute them in SP_i according to that order. The following example motivates this choice:

Example 6. *Consider the polynomial $P = a + 4abc - 2ad - 2adc$. Substituting a variable occurring k times in P by a polynomial containing h monomials increases the number of monomials in the result by $k \cdot (h-1)$ in the worst case. E.g. substituting $a = x + y + z + xz$ in P results in a new polynomial with 16 monomials. For this reason we refrain from substituting variables with large numbers of occurrences in P first. Instead we start with variables having smaller numbers of occurrences and hope that cancellation of monomials improves the situation when we finally arrive at the substitution of later variables. Assume in the example that we substitute $b = xy$ first, then $c = xz$ and then $d = xyz$, leading to $P = a$. After substituting $a = x + y + z + xz$ we arrive at the final result of the series of substitutions without exceeding a number of four monomials in between.*

In our experiments, we observed that the order mentioned above successfully keeps the sizes of intermediate polynomials small in many cases. Nevertheless, this method is only heuristical and may fail. Therefore, before each substitution we make a copy of SP_i (Line 7) to which we may backtrack if the size of the polynomial grows too much. Altogether, this leads to a *dynamic* substitution order based on the growth behavior we actually observe.

Example 7. *Consider the polynomial $P = abx + aby - 2abxy + ab + a$. Substituting the 4 occurrences of b by $m + n - mn$ leads to a polynomial with 13 monomials. After substituting $a = xy$ in the resulting polynomial we arrive at 4 monomials. However, if we substitute a (with one occurrence more than b) first, we obtain 2 monomials after the first substitution and again 4 monomials after substituting b. Of course, we prefer the second order which we obtain if we discard the substitution of b due to the size of the intermediate result.*

In Algorithm 2, this idea is implemented as follows: After making a copy of SP_i (Line 7), the polynomial for the first component in the sorted candidate list is substituted in SP_i (Line 10). We check the increase in size of the polynomials after substitution. To avoid a sharp increase in the number of monomials, we set the increase threshold to 10% (Line 8). If the increase is less than the threshold, we recognize it as a successful substitution. Thus, the candidate is removed from the list of components (Line 12 – Line 13), and the process is repeated by finding a new set of candidates (Line 3). On the other hand, if the increase is more than the threshold, then SP_i is restored to its state before substitution, and the process continues by substituting the next candidate in the sorted list (Line 15). If there is no substitution of any candidate that satisfies the threshold limit, the value of the threshold is multiplied by two and the process is repeated from the first candidate in the sorted list (Line 16 – Line 17).

In the next section we present the experimental results.

V. Experimental Results

We have implemented the proposed approach DyPoSub in C++. The experiments have been carried out on an Intel Xeon E3-1270 v3 with 3.50 GHz and 32 GByte of main memory. First, we have evaluated our approach on a wide range of optimized multipliers. The results are summarized in Table I. Please note that the *Time-Out* (TO) has been set to 24 hours. The first column *Size* of Table I shows the size of the multiplier based on the input bits. The second column *Benchmark* lists the architecture of the multiplier based on its stages (abbreviations are given below the table). These multipliers have been generated using the Arithmetic Module Generator [20] and GenMul [21]. The third column *Optimiz.* reports the performed optimization: this is either none (-), or the well-known abc [19] optimizations *dc2* and *resyn3*, respectively.

The verification data of our proposed approach is reported in the fourth column *Verification data* which consists of three subcolumns: *Nodes* shows the number of AIG nodes of the multiplier, *Vanishing Monomials* gives the total number of removed vanishing monomials, and *Max Poly Size* shows the maximum size of the current polynomial SP_i during backward rewriting by counting the number of monomials.

The fifth column *Ours* of Table I reports the overall run-time of our proposed approach in seconds. Finally, the run-times of the state-of-the-art verification methods are shown in the sixth column which consists of seven subcolumns. While the first subcolumn *Com.* reports the run-times of the commercial verification tool from Onespin, the remaining subcolumns give the run-times of the most recent available SCA approaches.

As can be seen, our approach is able to verify all non-optimized multipliers and only produces a time out for two benchmarks with optimization ($SP{\circ}BD{\circ}KS$ and $SP{\circ}WT{\circ}CL$; 4 instances in total). In contrast, none of the state-of-the-art SCA methods can verify any of the optimized multipliers. The commercial tool can prove one multiplier type incl. the optim. version, however only the 16×16 instance.

For one multiplier example we also demonstrate the effect of the proposed dynamic ordering in more detail. Fig. 5 shows the number of monomials (vertical axis) in each step of backward rewriting (horizontal axis) for the 32-bit $SP{\circ}DT{\circ}LF$ multiplier with and without optimization. The black line represents the use of a static ordering and the red line shows the polynomial sizes when using the proposed dynamic backward rewriting. Before optimization, both static and dynamic ordering allow to verify the circuit (black and red lines overlap from approx substitution step 300). However, when looking at both optimized multipliers the static ordering leads to (several) peaks with the consequence that the verification fails (see Fig. 5b and Fig. 5c). In contrast, as can be seen in all three figures the proposed dynamic ordering keeps the maximum size of SP_i small (orders of magnitudes smaller in comparison to the static order). Therefore, the proposed approach successfully verifies both optimized multipliers.

In a second experiment, we consider industrial benchmarks from the Synopsys DesignWare Library. They have been optimized for delay. The gate-level Verilog description of these multipliers is generated by mapping the multiplier IP to a standard cell library consisting of up to 3-input logical gates using Synopsys Design Compiler. Then, the Verilog description is converted to AIG using abc. The results can be found in Table II. In addition, the table also includes the highly optimized multiplier from the EPFL combinational benchmark suite [22]. As can be seen, the proposed approach is able to prove the correctness for all these multipliers while the commercial tool can only verify the smallest instance and all other SCA methods fail.

TABLE I

RESULTS OF VERIFYING OPTIMIZED MULTIPLIERS

Size	Benchmark	Optimiz.	Verification data			Run-times (seconds)							
			Nodes	Vanishing Monomials	Max Poly Size	Ours	Com.	[13]	[10]	[6]	[5]	[11]	[8]
16×16	$SP{\circ}DT{\circ}LF$	-	2,884	728	431	0.13	43.00	0.27	0.58	3.21	TO	TO	TO
		dc2	2,115	27	318	0.04	40.00	TO	TO	TO	TO	TO	TO
		resyn3	2,702	1,120	433	0.06	43.00	TO	TO	TO	TO	TO	TO
64×64	$SP{\circ}DT{\circ}LF$	-	48,808	2,249	7,256	7.22	TO	40.72	120.76	2,273.64	TO	TO	TO
		dc2	36,365	5,618	8,734	16.82	TO	TO	TO	TO	TO	TO	TO
		resyn3	45,115	3,038	6,908	7.58	TO	TO	TO	TO	TO	TO	TO
	$SP{\circ}AR{\circ}CK$	-	48,073	0	4,284	5.33	TO	151.17	TO	TO	TO	TO	TO
		dc2	36,154	0	4,285	5.30	TO	TO	TO	TO	TO	TO	TO
		resyn3	43,501	0	4,284	5.41	TO	TO	TO	TO	TO	TO	TO
	$SP{\circ}BD{\circ}KS$	-	50,756	613,454	5,607	14.30	TO	162.26	TO	TO	TO	TO	TO
		dc2	37,655	-	-	TO	TO	TO	TO	TO	TO	TO	TO
		resyn3	47,746	-	-	TO	TO	TO	TO	TO	TO	TO	TO
	$SP{\circ}WT{\circ}CL$	-	68,875	266,684	4,461	19.76	TO	96.27	224.43	TO	TO	TO	TO
		dc2	44,643	-	-	TO	TO	TO	TO	TO	TO	TO	TO
		resyn3	63,962	-	-	TO	TO	TO	TO	TO	TO	TO	TO
	$BP{\circ}AR{\circ}RC$	-	38,439	0	20,099	21.50	TO	78.61	70.43	911.07	0.09	TO	TO
		dc2	31,312	0	15,882	21.43	TO	TO	TO	TO	TO	TO	TO
		resyn3	34,317	0	20,097	23.21	TO	TO	TO	TO	TO	TO	TO
	$BP{\circ}OS{\circ}CU$	-	39,798	0	25,803	34.68	TO	302.01	TO	TO	TO	TO	TO
		dc2	31,925	0	26,200	50.78	TO	TO	TO	TO	TO	TO	TO
		resyn3	37,156	0	20,100	27.12	TO	TO	TO	TO	TO	TO	TO
128×128	$SP{\circ}AR{\circ}RC$	-	162,304	0	16,640	108.82	TO	966.57	TO	TO	1.10	TO	TO
		dc2	146,044	0	16,642	101.63	TO	TO	TO	TO	TO	TO	TO
		resyn3	162,298	0	16,640	104.64	TO	TO	TO	TO	TO	TO	TO
	$SP{\circ}DT{\circ}LF$	-	164,572	3,642	29,811	194.09	TO	527.37	TO	TO	TO	TO	TO
		dc2	146,655	43,086	57,708	1905.57	TO	TO	TO	TO	TO	TO	TO
		resyn3	150,711	4,900	28,721	196.68	TO	TO	TO	TO	TO	TO	TO
	$SP{\circ}WT{\circ}BK$	-	166,938	1,623	22,406	172.14	TO	706.34	TO	TO	TO	TO	TO
		dc2	149,353	4,160	26,911	329.81	TO	TO	TO	TO	TO	TO	TO
		resyn3	154,146	1,808	22,347	161.30	TO	TO	TO	TO	TO	TO	TO

Stage 1 ⇒ **SP**: Simple partial product generator **BP**: Booth partial product generator TO: Time-Out

Stage 2 ⇒ **AR**: Array **DT**: Dadda tree **WT**: Wallace tree **OS**: Overturned-stairs tree **BD**: Balanced delay tree

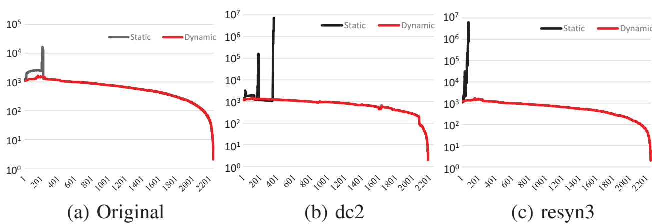
Stage 3 ⇒ **RC**: Ripple carry **BK**: Brent-Kung **LF**: Ladner-Fischer **CK**: Carry-skip **CU**: Conditional sum **CL**: Carry look-ahead **KS**: Kogge-Stone

TABLE II

RESULTS OF VERIFYING INDUSTRIAL MULTIPLIERS

Source	Size	Nodes	Run-times (seconds)							
			Ours	Com.	[13]	[10]	[6]	[5]	[11]	[8]
Synopsys DesignWare Library (pparch*)	16 × 16	2,432	0.13	40	TO	TO	TO	TO	TO	TO
	32 × 32	7,240	2.27	TO	TO	TO	TO	TO	TO	TO
	48 × 48	16,086	16.97	TO	TO	TO	TO	TO	TO	TO
	64 × 64	27,658	62.69	TO	TO	TO	TO	TO	TO	TO
	96 × 96	61,180	506.86	TO	TO	TO	TO	TO	TO	TO
	128×128	106,949	1,861.56	TO	TO	TO	TO	TO	TO	TO
	160×160	166,492	4,569.96	TO	TO	TO	TO	TO	TO	TO
	192×192	238,920	9,846.22	TO	TO	TO	TO	TO	TO	TO
	256×256	422,077	29,988.20	TO	TO	TO	TO	TO	TO	TO
EPFL mul.	64 × 64	27,190	76.89	TO	TO	TO	TO	TO	TO	TO

*Delay-optimized flexible Booth Wallace after technology mapping



(a) Original (b) dc2 (c) resyn3

Fig. 5. SP_i size during backward rewriting of 32-bit $SP \circ DT \circ LF$ using static (black line) and dynamic (red line) orderings

VI. CONCLUSION

In this paper, we have proposed our novel SCA-based verification approach DYPOSUB. For the first time a dynamic substitution order is used during backward rewriting which allows to control the size of the intermediate polynomial. Our approach allows to verify optimized and technology mapped multipliers including industrial benchmarks which has not been possible before.

REFERENCES

[1] 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.

[2] R. P. Kurshan and L. Lamport, "Verification of a multiplier: 64 bits and beyond," in *CAV*, 1993, pp. 166–179.

[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] 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.

[5] C. Yu, M. Ciesielski, and A. Mishchenko, "Fast algebraic rewriting based on and-inverter graphs," *TCAD*, vol. 37, no. 9, pp. 1907–1911, 2017.

[6] 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.

[7] 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.

[8] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD*, 2017, pp. 23–30.

[9] A. Mahzoon, D. Große, and R. Drechsler, "Combining symbolic computer algebra and boolean satisfiability for automatic debugging and fixing of complex multipliers," in *ISVLSI*, 2018, pp. 351–356.

[10] 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.

[11] D. Ritirc, A. Biere, and M. Kauers, "Improving and extending the algebraic approach for verifying gate-level multipliers," in *DATE*, 2018, pp. 1556–1561.

[12] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *FMCAD*, 2019, pp. 28–36.

[13] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *DAC*, 2019, pp. 185:1–185:6.

[14] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for lut-based fpgas," in *FPGAs for Custom Computing Machines*, 1998, pp. 35–42.

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

[16] D. Kaufmann, A. Biere, and M. Kauers, "Incremental column-wise verification of arithmetic circuits using computer algebra," *Formal Methods in Sys. Design*, Feb. 2019.

[17] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. A. K. Peters, Ltd., 2001.

[18] R. Zimmermann, "Binary adder architectures for cell-based vlsi and their synthesis," Ph.D. dissertation, Swiss Federal Institute of Technology, 1997.

[19] "Abc: A system for sequential synthesis and verification," available at https://people.eecs.berkeley.edu/~alanmi/abc/, 2018.

[20] "Arithmetic module generator based on acg," available at https://www.ecsis.riec.tohoku.ac.jp/topics/amg/i-amg, 2019.

[21] A. Mahzoon, D. Große, and R. Drechsler, "GenMul: Generating architecturally complex multipliers to challenge formal verification tools," in *IWLS*, 2019.

[22] L. Amaru, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *IWLS*, 2015.