# A Conflict-Aware Learning Approach to SCA Verification for MAC Architectures

Jan Kleinekathöfer[U], Lennart Weingarten[U], Kamalika Datta[Δ,U], Rolf Drechsler[Δ,U]
[Δ]*German Research Centre for Artificial Intelligence (DFKI), Bremen, Germany*
[U]*Institute of Computer Science, University of Bremen, Germany*
{ja_kl, len_wei, kdatta, drechsler}@uni-bremen.de

***Abstract*—Digital circuit verification remains an important area of research. To address this, various proof engines like Boolean Satisfiability (SAT), Binary Decision Diagrams (BDD), Answer Set Programming (ASP), and Symbolic Computer Algebra (SCA) have been explored. For specific complex circuits such as multipliers, Multiply and Accumulate (MAC) units, and Dot-Products, SCA has shown promise in the verification process. However, a major issue with SCA is the intermediate term explosion that can occur during verification.**

**In this work, we design an SCA-based proof engine that exploits a conflict-aware learning approach. Together with dynamic ordering strategies and node polarity this compresses the intermediate polynomial expression, which significantly reduces the problem of term explosion. The approach introduces a preprocessing step that identifies and learns conflicts from the circuit structure and applies this knowledge during the verification to reduce polynomial size. We demonstrate the importance of each technique and specifically show how our proposed conflict optimization impacts the verification outcome. We experimentally validate the effectiveness of this approach by comparing our results with a state-of-the-art SCA-based proof engine using optimized MAC and other MAC architectures. The results demonstrate that the proposed method successfully verifies circuits that remained unverifiable using previous approaches.**

***Index Terms*—Symbolic Computer Algebra (SCA), Circuit verification, Polarity, Dynamic Substitution, Conflict Optimization, Multiply–Accumulate (MAC)**

## I. Introduction

As the complexity of digital systems increases, the risk of design errors and functional bugs surge drastically. Traditional verification methods, such as simulation, are often insufficient because they fail to guarantee complete coverage. This has motivated the need for more rigorous, formal verification techniques that provide mathematical proofs of correctness.

To address this, various proof engines like *Boolean Satisfiability* (SAT) [1], *Binary Decision Diagrams* (BDD) [2], *Answer Set Programming* (ASP), and *Symbolic Computer Algebra* (SCA) [3]–[8] have been explored. Each of these techniques has its own strengths and weaknesses. SAT solvers are powerful for handling propositional logic problems but can struggle with complex arithmetic. BDDs are excellent for representing and manipulating Boolean functions efficiently, yet their size can explode for certain circuit types specially multipliers. ASP offers a high-level, declarative approach, but can be computationally expensive for multipliers. For specific complex circuits such as multipliers, *Multiply and Accumulate* (MAC) units, and *Dot-Products* (DP), SCA has shown promise in the verification process.

To this end, several SCA-based techniques have been developed [3], [4], [6]–[17]. Although many works have extensively explored multiplier verification using SCA, very few have explored MAC and DPs [16], [18], [19]. In a recent work [6], certain phase selection and variable ordering are used to optimize SCA-based verification, offering heuristics that minimize polynomial growth and enhance performance for multipliers compared to existing methods. However, these techniques have not been studied for MAC units.

In this paper, we introduce conflict optimization as a two-step technique for compressing intermediate polynomials. The first step is the conflict analysis which extracts conflicts from the circuit structure prior to verification. The second step is the conflict removal exploiting conflicts during verification to eliminate monomials that would eventually vanish. Combined with dynamic variable ordering and node polarity (phase) optimization, this approach significantly reduces the size of intermediate polynomial expressions for MAC units.

Following are the main contribution of this paper:

1) Conflict optimization is introduced as a new learning-based approach for intermediate polynomial compression.
2) An improved SCA-based proof engine is designed, incorporating conflict optimization, dynamic ordering, and polarity strategies.
3) The impact of advanced techniques on the verifiability of MAC designs is analyzed.
4) Extensive experimentation has been performed on simple and optimized MAC designs.

## II. Background

The SCA verification methodology begins with the definition of the *Specification Polynomial* (SP), proceeds through the identification of *Gate Polynomials* (GP) or *Node Polynomials* (NP), and finally concludes with a *backward rewriting* process.

This technique is versatile, supporting circuits represented as gate-level netlists or *And-Inverter Graphs* (AIGs). The process initiates by constructing the SP, which mathematically models the circuit's functionality. Next, a set of GPs or NPs is derived for every gate or node within the design. The verification core is the backward rewriting phase: starting from the circuit outputs and moving in reverse topological order, the SP is iteratively updated by substituting node variables with their corresponding polynomials. Once the process reaches the primary inputs, the resulting expression is evaluated. A

remainder of zero confirms functional correctness, whereas any non-zero remainder signals a design fault.

For an AIG, the relationship between a node's output edge and its inputs is defined by specific gate polynomial rules, summarized below:

TABLE I: And-Inverter Graph Substitution Rules

| Nr. | Rule | Description |
|---|---|---|
| R1 | $z = a$ | The output edge functions as a buffer or direct wire |
| R2 | $z = 1 - a$ | The output edge represents a complemented (inverted) signal |
| R3 | $z = ab$ | A standard AND gate with no complemented inputs |
| R4 | AND gate with a single complemented input | |
| R4a | $z = b - ab$ | (input $a$ is complemented) |
| R4b | $z = a - ab$ | (input $b$ is complemented) |
| R5 | $z = 1 - a - b + ab$ | An AND gate where both inputs are complemented |

### A. Example Evaluation using SCA for Majority Operation

Fig. 1 illustrates the AIG for a *Majority* (MAJ) function. Given primary inputs $a, b$, and $c$, and output $out$, the SP is defined as:

$$SP_{MAJ} = out + 2abc - ab - ac - bc = 0$$

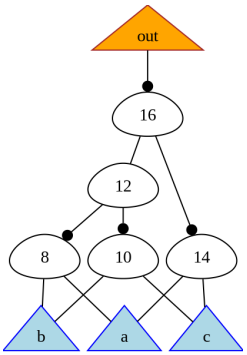



Fig. 1: AIG Representation of a Majority Function. Triangles denote Primary I/O, Ovals represent AND gates, and Black Circles indicate Complemented Edges.

The verification of the majority function (Fig. 1) is performed via backward rewriting, initiated from the primary output. The process comprises six substitution steps: five corresponding to the internal AIG nodes and one for the output node. Each step applies the algebraic transformation rules defined in Table I, determined by the respective node type. After the output is processed in $P_0$, each node is processed according to its node ID, from highest to lowest, with its corresponding gate polynomial rule. The polynomial at each step $i$ is marked with $P_i$, while the node and rule applied in step $i$ are denoted by $\xrightarrow[\text{nodeID}]{\text{rule}}$.

The nodes are replaced by their corresponding gate polynomial rules, and the expression is simplified according to general pseudo-boolean and algebraic rules. It is observed, that the polynomial reaches its maximum size following the substitution step for $P_3$. Even for a small 3-bit function, the polynomial grows from an initial size of 5 to a maximum of 9 terms. This indicates that verifying a MAJ function requires more than double the initial memory needed to generate the specification polynomial during the substitution process.

$$\begin{aligned}
P_0 \xrightarrow[out]{R2} &= (out) + 2abc - ab - ac - bc \\
&= 1 - n16 + 2abc - ab - ac - bc \\
P_1 \xrightarrow[n16]{R4b} &= 1 - (n16) + 2abc - ab - ac - bc \\
&= 1 - (n12 - n12n14) + 2abc - ab - ac - bc \\
&= 1 - n12 + n12n14 + 2abc - ab - ac - bc \\
P_2 \xrightarrow[n14]{R3} &= 1 - n12 + n12(n14) + 2abc - ab - ac - bc \\
&= 1 - n12 + n12ac + 2abc - ab - ac - bc \\
P_3 \xrightarrow[n12]{R5} &= 1 - (n12) + (n12)ac + 2abc - ab - ac - bc \\
&= n8 + n10 - n8n10 - n8ac - n10ac \\
&\quad + n8n10ac + 2abc - ab - bc \\
P_4 \xrightarrow[n10]{R3} &= n8 + (n10) - n8(n10) - n8ac - (n10)ac + n8(n10)ac \\
&\quad + 2abc - ab - bc \\
&= n8 - n8bc - n8ac + n8abc + abc - ab \\
P_5 \xrightarrow[n8]{R3} &= (n8) - (n8)bc - (n8)ac + (n8)abc + abc - ab \\
&= (+ab - ab) - 2abc + 2abc = 0
\end{aligned}$$

Fig. 2: Substitution Steps for a Majority Function

### B. Related Work

Over the last decade, SCA has emerged as a powerful framework for the formal verification of complex arithmetic circuits, including multipliers, dividers, and *Multiply-Accumulate* (MAC) units [3], [4], [6]–[16]. The majority of these approaches operate directly on AIG representations [5], [15], [16], [18], [20].

Researchers have explored diverse strategies to mitigate the polynomial explosion problem inherent in SCA. Proposed solutions range from redundant term elimination and algebraic simplification to column-wise decomposition methods that partition the verification task into more manageable sub-problems [4]. While reverse engineering can extract structural and algebraic relationships to aid in the verification of optimized designs [5], the approach is often computationally intensive. Recent heuristics for variable ordering and phase selection [6], along with hybrid SCA-SAT solvers [3], [7], have further optimized performance. Additionally, the computational limits of Gröbner basis rewriting have been addressed through parallelization and memory optimization [8].

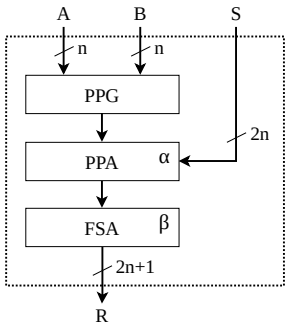
Despite these developments, the backward rewriting process remains susceptible to increasing intermediate polynomial complexity, leading to excessive memory usage and runtime

even for smaller MAC circuits [19]. This challenge is exacerbated in optimized circuits where the underlying structure is unknown, resulting in significant verification overhead [19]. In this work, we focus on understanding the effect of some advanced techniques on verifiability particularly for MAC circuits. Analysis is conducted to identify the correct signal polarity while bypassing the need for exhaustive search. Finally, the application of conflict analysis is shown to mitigate intermediate polynomial blow-up, enabling the verification of circuits that were previously not possible for existing methods [18], [19].

## III. MAC Generation

This section provides the details of the MAC generation process and of our tool `MAC-Gen` which generates a variety of structurally different MAC architectures. The left-hand side of Fig. 3 shows the general architecture of the MAC unit including the notation of the bit-width: The bit-level description of the MAC is defined as

$$R_{2n+1} = (A_n \times B_n) + S_{2n} \tag{1}$$




| PPG | |
|---|---|
| 1 | [U] Simple Unsigned |
| **PPA** | |
| 1 | [AR] Array |
| 2 | [CWT] Counter based Wallace Tree |
| 3 | [DT] Dadda Tree |
| 4 | [WT] Wallace Tree |
| **FSA** | |
| 1 | [RC] Ripple Carry |
| 2 | [CL] Carry Look-Ahead |
| 3 | [LF] Ladner-Fischer |
| 4 | [KS] Kogge-Stone |
| 5 | [BK] Brent-Kung |
| 6 | [CS] Carry Skip |
| 7 | [SE] Serial Prefix |

Fig. 3: MAC Architecture Options

The *Multiply-Accumulate* (MAC) unit extends a *Multiplier Unit* (MUL) with an additional input $S$. The MUL consists of three stages: the *Partial Product Generator* (PPG), the *Partial Product Accumulator* (PPA), and the *Final Stage Adder* (FSA) [21].

The MUL processes two $n$-bit inputs, $A$ and $B$ and for the MAC there is an additional $2n$-bit input $S$. Rather than adding the third operand $S$ after the final product summation in the FSA, it is integrated directly into the PPA. The output of the PPG stage combined with the third operand $S$ forms the inputs of the PPA. This architectural choice shortens the critical path by eliminating the need for a dedicated late-stage accumulation adder. Consequently, this not only enhances the overall operating speed of the MAC unit but also yields significant savings in both power consumption and silicon area [22].

The right side of Fig. 3 presents the possible options for each component, i.e. for the PPA one out of four designs is chosen and for FSA one out of seven. Resulting in a total number of $4 \times 7 = 28$ different configurations. To compactly denote a MAC configuration consisting of a Dadda Tree (PPA) and Carry Skip adder (FSA) the following notation is used: DT_CS.

**Algorithm 1** Enhanced SCA-Based Verification with Integrated Dynamic Evaluation

**Require:** Gate-level circuit $C = (V, E)$, output polynomial $P_{out}$, growth threshold $\tau$
**Ensure:** Polynomial $P_{in}$ over primary inputs or verification failure
conflicts ← **ComputeConflicts**$(C)$
$P \leftarrow P_{out}$
$A \leftarrow$ primary outputs of $C$
**while** $A$ contains non-input signals **do**
  bestGate ← **null**
  bestGrowth ← $\infty$
  **for** each gate $g$ in available gates $A$ **do**
    $P' \leftarrow$ **Substitute**$(P, g)$
    $P' \leftarrow$ **ConflictPolarityOpt**$(P', g,$ conflicts$)$
    growth ← $|P'|/|P|$
    **if** growth $<= \tau$ **then**
      bestGate ← $g$
      **break**
    **else if** growth $<$ bestGrowth **then**
      bestGate ← $g$
      bestGrowth ← $growth$
    **end if**
  **end for**
  $P \leftarrow$ **Substitute**$(P,$ bestGate$)$
  $P \leftarrow$ **ConflictPolarityOpt**$(P,$ bestGate, conflicts$)$
  $A \leftarrow$ **UpdateAvailableGates**$(A,$ bestGate$)$
**end while**
**return** $P$

**Algorithm 2** ConflictPolarityOpt

**Require:** Polynomial $P$, gate $g$, conflict set $\mathcal{C}$
**Ensure:** Optimized polynomial $P^*$
$P^* \leftarrow$ **RemoveConflicts**$(P, \mathcal{C})$
**for** each input signal $i$ of $g$ **do**
  $P' \leftarrow$ **SwapPolarity**$(P^*, i)$
  $P' \leftarrow$ **RemoveConflicts**$(P', \mathcal{C})$
  **if** $|P'| < |P^*|$ **then**
    $P^* \leftarrow P'$
  **end if**
**end for**
**return** $P^*$

## IV. SCA-Based Verification Methodology

While the core algorithm of SCA-based verification, as presented in Section II, is conceptually simple, several extensions are required to achieve efficient verification in practice. Even when applying dynamic substitution ordering [5] as well as the polarity optimization [6] intermediate term explosion still prevents the verification of complex circuits. Therefore, we introduce conflict optimization as a new method for polynomial compression. It can be seen as a generalization of vanishing monomials introduced in [23]. In the preprocessing step, we perform conflict analysis to learn node dependencies. These insights are then applied during verification to remove monomials that contain conflicts, thereby reducing the polynomial expression. We combine this technique with dynamic

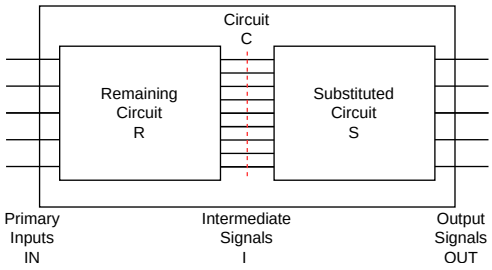



Fig. 4: Cut through the Design Under Verification during the SCA Substitution

substitution ordering and polarity optimization to create a powerful verification engine.

While others have also suggested grouping substitution steps into atomic blocks and/or cones [5], we present a purely gate-based approach. Although grouping gates into larger blocks can simplify processing, detecting such blocks so that they can be substituted in a single step, as described in [5], is often computationally expensive and challenging for circuits with unknown internal structures.

The overall approach is presented in Algorithms 1 and 2. In general, Algorithm 1 initializes the Polynomial $P$ with the output specification and then substitutes gates until the primary inputs are reached. Available gates are stored in $A$. Gates become available when all successors are substituted. In the substitution loop, the next gate is determined dynamically and after each substitution, polarity and conflict optimization are applied, which is extracted to Algorithm 2. In the remainder of this section, we first briefly introduce the dynamic substitution ordering and polarity optimization before detailing the concept of the conflict optimization.

### A. Dynamic Substitution Ordering

At every step during the SCA verification, the polynomial represents the output function of the circuit depending on a set of signals. At the beginning, these signals are the primary outputs, during the verification, they are intermediate signals, and in the end, they are the input signals. This is illustrated in Fig. 4. At the beginning and at the end of the process, the set of signals is fixed and offers no flexibility. At these stages, the function either admits a compact polynomial representation or it does not. In contrast, the choice of intermediate signals offers a degree of flexibility. While the signals must always form a valid cut through the circuit, and in AIGs each step replaces one signal by its two inputs, there are multiple strategies for advancing this cut.

Although many static substitution orderings are possible (e.g., breadth-first or depth-first), prior work has shown that dynamically selecting substitutions based on their impact on polynomial size yields significantly better results. To control polynomial growth, we adopt a dynamic substitution strategy inspired by [24]. For each gate whose output currently belongs to the cut, a tentative substitution is applied and the resulting change in polynomial size is determined. The loop is immediately stopped if one substitution is found which does not exceed the growth limit (set to 1.3 in our experiments) Based on our observations, a factor of 1.3 provides an good balance by allowing a 30% margin for the engine's expansion while maintaining the polynomial size within a manageable growth limit. If all candidates exceed the predefined growth limit, the substitution with the smallest relative increase is selected. The corresponding procedure is summarized in Algorithm 1 (lines 5–18). Although this approach requires repeated trial substitutions, it effectively prevents blow-up of intermediate representations and consistently leads to shorter overall verification runtimes.

### B. Polarity Optimization

Further polynomial size reduction is achieved through polarity optimization, as introduced in [6]. During SCA substitution, each polynomial represents the output function with respect to a cut set $I$ of signals. In some cases, intermediate polynomials grow excessively large; polarity optimization mitigates this effect by negating selected signals to reduce the number of monomials. As a representative example, consider the three-input NOR function, which evaluates to zero if one or more of its inputs are one. When expressed directly in terms of the positive-polarity signals $a$, $b$, and $c$, its arithmetic polynomial contains multiple higher-degree terms, $\mathrm{NOR}(a,b,c) = 1 - a - b - c + ab + ac + bc - 2abc$. In contrast, allowing the inputs to appear in complemented form yields a significantly more compact representation, $\mathrm{NOR}(a,b,c) = \bar{a}\bar{b}\bar{c}$. This effect is analogous to the distinction between Positive Polarity Reed–Muller expressions and Fixed Polarity Reed–Muller expressions at the Boolean level [25]. To maintain canonical form, each signal must appear with a single polarity. Polarity optimization significantly reduces polynomial size, surpassing state-of-the-art methods and making previously unverifiable designs verifiable [6]. As presented in Algorithm 2, polarity assignments are determined dynamically: When substituting a signal by its gate function, the look ahead algorithm evaluates whether flipping the signal's polarity decreases the polynomial size for each signal in the gate function. Note that this heuristic does not guarantee an optimal polarity assignment, as exhaustively exploring all combinations is infeasible; nevertheless, it has a substantial impact on polynomial size and, consequently, on verifiability.

### C. Conflict Optimization

Since SCA proceeds from the circuit outputs toward the inputs, the relationships between signals in the remaining, unsubstituted part of the circuit are unknown at intermediate steps. This often leads to unnecessary complex intermediate polynomials.

This problem can be formalized as follows: Consider a circuit $C$ with a set of inputs $IN$ and a set of outputs $OUT$ which represents the function $f_C : IN \rightarrow OUT$. Moreover, let $I$ be a set of intermediate signals which constitute a cut through the circuit. The circuit function $f_C$ is now split into two subfunctions: The function $f_S : I \rightarrow OUT$ from the intermediate signals to the primary outputs and the function $f_R : IN \rightarrow I$ from the primary inputs to the intermediate signals. The function $f_R$ is not necessarily injective. In this

**Algorithm 3** ComputeConflicts - Conflict Computation by Implication Propagation

**Require:** AIG $g = (V, E)$
**Ensure:** Set of polarity-aware conflicts $\mathcal{C}$

```
1: C ← ∅
2: for each signal s ∈ V do
3:    for each value v ∈ {0,1} do
4:       assignments ← ∅
5:       queue ← empty
6:       enqueue(queue, s)
7:       insert(assignments, (s, v))
8:       /* Calculate implications */
9:       while queue is not empty do
10:         t ← dequeue(queue)
11:         for each implication (u, w) induced by t in g do
12:            if not contains(assignments,u) then
13:               enqueue(queue, u)
14:               insert(assignments, (u, w))
15:            end if
16:         end for
17:      end while
18:      /* Extract conflicts */
19:      for each (u, w) ∈ assignments do
20:         if u ≠ s then
21:            add conflict ((s, v), (u, w)) to C
22:         end if
23:      end for
24:   end for
25: end for
26: return C
```

case, there exist assignments to $I$ that cannot occur and therefore do not need to be represented by $f_S$. Not mapping these input assignments can significantly simplify $f_S$.

This behavior was already noticed in PolyCleaner [23]. There it was specifically attributed to the outputs of half adders. No matter the half adder inputs, the sum and carry output will never be one at the same time. Therefore, each monomial containing the sum and the carry signal at the same time will vanish to zero after substitution of the half adder gates. Thus, these monomials are called vanishing monomials and it was shown that they have huge impact on the intermediate polynomial size.

We generalize this concept beyond half-adder outputs to arbitrary pairs of mutually exclusive signals. We call the exclusivity of two signals a *conflict*. Each monomial containing both signals from a conflict is a vanishing monomial. We refer to the process of identifying conflicts during the preprocessing step as conflict analysis. Since we also utilize polarity optimization, these conflicts possess polarity as well. So assuming signal $a$ and $b$ it could be that the $a$ with positive polarity conflicts $b$ with negative polarity. Note, that conflicts are symmetric: If $a$ conflicts $b$, $b$ conflicts $a$ (with respective polarities).

Our approach for the conflict analysis is presented in Algorithm 3. The algorithm iterates over all signals in the graph and considers both constant assignments for each signal. For each signal value combination $(s, v)$, the implications are calculated, and the resulting conflicts are extracted. The assignment data structure is implemented using a hash map. Additionally, a queue is used to store signals for which implications must be processed. The implication rules for

TABLE II: Implication Rules

(a) Forward Implication

| Input 1 | Input 2 | Output |
|---|---|---|
| 0 | x | **0 (I)** |
| x | 0 | **0 (I)** |
| 1 | 1 | **1 (I)** |

(b) Backward Implication

| Input 1 | Input 2 | Output |
|---|---|---|
| **1 (I)** | **1 (I)** | 1 |
| **0 (I)** | 1 | 0 |
| 1 | **0 (I)** | 0 |

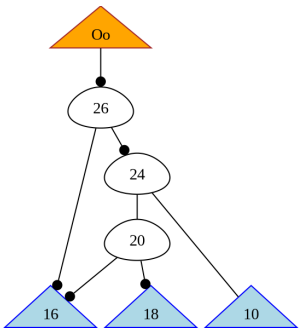



Fig. 5: Example Circuit for Conflict Analysis

AIGs are presented in TABLE II. Implied values are marked with (I). The signal at the output of an AND gate can be set if either one of the inputs is assigned 0 or both inputs are assigned to 1. We call this *forward implication*. If the signal at the output of a gate is assigned 1, the inputs can also be assigned 1. When the output is assigned 0 and one of the inputs is assigned 1 the other one can be implied to be 0. We call this *backward implication*. In practice, complemented edges introduce additional cases that slightly complicate these rules. After all possible implications have been propagated, conflicts are extracted from the resulting assignments.

After the conflicts were learned by the conflict analysis, conflict removal has to be performed as the second step during the verification to achieve a polynomial size reduction. Algorithm 1 and 2 outline how the conflict removal can be integrated into the SCA substitution. Every time a signal is substituted, or a polarity is flipped, the polynomial is checked for monomials that contain conflicts. If a monomial contains a conflict, it is removed.

In the following, we demonstrate the conflict analysis and its effect on SCA substitution using the example circuit from Fig. 5. This circuit was extracted from a prefix adder. Table III shows all implications that can be calculated in this graph. The first column presents the initial assumption in the format 'a:1/0', where 'a' is the node ID and '1/0' are the constant values assigned to that node. The remaining columns present the implied values for all nodes in the format 'c/d', where

TABLE III: Implications for the Circuit from Fig. 5

| Assumption | 10 | 16 | 18 | 20 | 24 | 26 |
|---|---|---|---|---|---|---|
| 10:1/0 | 1/0 | x/x | x/x | x/x | x/0 | x/x |
| 16:1/0 | x/x | 1/0 | x/x | 0/x | 0/x | 0/x |
| 18:1/0 | x/x | x/x | 1/0 | 0/x | 0/x | x/x |
| 20:1/0 | x/x | 0/x | 0/x | 1/0 | x/0 | x/x |
| 24:1/0 | 1/x | 0/x | 0/x | 1/x | 1/0 | 0/x |
| 26:1/0 | x/x | 0/x | x/x | x/x | 0/x | 1/0 |

'c' is the implied value for 1 and 'd' the implied value for 0; 'x' represents an unknown value. The table shows that, for some assumptions, no further implications can be derived. If node 10 is set to 1 no other nodes can be implied, as node 10 only is a non-negated input to gate 24 and the constant value 1 does not control the output value of 24. Other assumptions like value 1 at node 16 lead to more implications. A 1 at node 16 directly implies a 0 at node 26 and 20. The 0 at node 20 implies a 0 at node 24. An implication of all values like presented for a 1 at node 24 usually does not occur in larger graphs. For larger graphs, the implication table is significantly sparse; consequently, unknown values are not stored in practice. Conflicts can directly be derived from this table. For simplicity, we do not consider polarity optimization in this example and assume all signals are used in positive polarity. Under this assumption, a *conflict* is present between the signal of the assumption and an implied signal if the implied signal has the opposite value with respect to the assumption. Exemplarily, for the assumption 16:1 the values 20:0, 24:0 and 26:0 are implied. Therefore, signal 16 is in conflict with signal 20, 24 and 26 and every monomial containing signal 16 together with one of the three will vanish to 0 and therefore can be removed immediately.

Fig. 6 presents how these conflicts can be used during SCA substitution to reduce the size of intermediate polynomial. In the first step, the output is replaced by the negation of signal $n26$ using rule 2. Please refer Table I for substitution rules. Following, signal $n26$ is replaced using rule 5. This results in the polynomial $n24+n16+n16n24$. The monomial $n16n24$ is highlighted in red as it contains a conflict and can therefore be removed reducing the polynomial to $n24+n16$. Afterward, $n24$ and $n20$ are substituted using rule 3 and 5 respectively. The final polynomial would have been identical without conflict removal, but intermediate monomials would have been larger. The quantitative impact of conflict analysis is evaluated in the experimental section.

$$
\begin{aligned}
SP_0 \xrightarrow[o0]{r2} &= 1 - n26 \\
SP_1 \xrightarrow[n26]{r5} &= 1 - (1 - n24 - n16 + n16n24) \\
&= n24 + n16 - n16n24 \\
&= n24 + n16 \\
SP_2 \xrightarrow[n24]{r3} &= n20n10 + n16 \\
SP_3 \xrightarrow[n20]{r5} &= (1 - n16 - n18 + n16n18)n10 + n16 \\
&= n10 - n16n10 - n18n10 + n16n18n10 + n16
\end{aligned}
$$

Fig. 6: Substitution Steps of the Circuit from Fig. 5 using Conflict Analysis

## V. Experiments

The SCA-based proof engine is implemented using C++ and all the experiments presented, are performed on a AMD Ryzen 7 PRO 4750U with 40GB main memory. For all experiments, a 30-minute timeout was set. Two types of MAC designs were experimentally evaluated: the structurally defined MACs, as described in Section III, and behavioral MACs, which are described here. The structural MACs described in this work differ from those in the reference work [18]. In [18], a four-stage architecture is used for the MAC; however, the MAC architecture in this work is more optimized, as an additional adder stage for the third MAC operand is not required. The behavioral MAC refers to a design defined at a high-level behavioral Verilog [19], which is then synthesized using a design compiler. For the generation the commands in yosys: `synth`, `opt` and abc: `strash` and `refactor` are used. In this work, Yosys and ABC were used to synthesize the Verilog code into an *And-Inverter Graph* (AIG) representation. The specific architecture and level of optimization are determined by the design compiler and remain unknown to the user.

To evaluate the effectiveness of the proposed dynamic substitution ordering with phase and conflict extension, 28 structural MAC designs were experimentally evaluated for 8, 16, and 32-bit widths. The results are presented in Table IV. From left to right, the table is structured as follows: first, the name of the MAC configuration is provided, followed by three groups—one each for 8, 16, and 32-bit. Each group consists of the results for the *Maximum Polynomial* (MP) size and the *Verification Time* (VT) in seconds. From top to bottom, the designs are grouped according to the PPA: Array (AR), Counter-based Wallace Tree (CWT), Dadda Tree (DT), and

TABLE IV: Verification Results for Structural MAC with Dynamic Substitution, Phase and Conflict Extensions

| | 8-Bit | | 16-Bit | | 32-Bit | |
|---|---|---|---|---|---|---|
| Design | MP | VT | MP | VT | MP | VT |
| AR_BK | 181 | 0.0395 | 757 | 0.5583 | 14013 | 77.1151 |
| AR_CL | 637 | 0.2928 | 5874 | 21.1273 | T.O. | T.O. |
| AR_CS | 420 | 0.0681 | 522 | 0.4039 | 100194 | 198.9790 |
| AR_KS | 172 | 0.0438 | 1380 | 1.5311 | 5672 | 97.5673 |
| AR_LF | 170 | 0.0413 | 975 | 0.8538 | 40830 | 135.3790 |
| AR_RC | 170 | 0.0436 | 543 | 0.3768 | 1837 | 5.9557 |
| AR_SE | 191 | 0.0428 | 544 | 0.3768 | 1901 | 5.8890 |
| CWT_BK | 155 | 0.0461 | 2540 | 2.7147 | 12168 | 25.1730 |
| CWT_CL | 3444 | 1.1193 | T.O. | T.O. | T.O. | T.O. |
| CWT_CS | 189 | 0.0569 | 3614 | 1.7097 | T.O. | T.O. |
| CWT_KS | 434 | 0.0650 | 3541 | 1.9208 | 17826 | 28.1186 |
| CWT_LF | 226 | 0.0478 | 1241 | 1.0529 | 75843 | 200.9170 |
| CWT_RC | 248 | 0.0514 | 18555 | 8.9134 | 109795 | 74.1498 |
| CWT_SE | 373 | 0.0593 | 67259 | 56.4319 | 53261 | 119.9360 |
| DT_BK | 240 | 0.0457 | 980 | 0.9581 | 13597 | 77.8349 |
| DT_CL | 815 | 0.5289 | T.O. | T.O. | T.O. | T.O. |
| DT_CS | 261 | 0.0607 | T.O. | T.O. | T.O. | T.O. |
| DT_KS | 476 | 0.0726 | 1943 | 1.1814 | 26984 | 148.7580 |
| DT_LF | 390 | 0.0804 | 834 | 0.9290 | 6159 | 37.9780 |
| DT_RC | 182 | 0.0440 | 678 | 0.4853 | 2179 | 7.3185 |
| DT_SE | 176 | 0.0408 | 666 | 0.4342 | 2244 | 7.3054 |
| WT_BK | 169 | 0.0417 | 875 | 0.7604 | 3223 | 13.2759 |
| WT_CL | 4862 | 4.7639 | 595692 | 1334.5800 | T.O. | T.O. |
| WT_CS | 246 | 0.0553 | 2166 | 1.1536 | T.O. | T.O. |
| WT_KS | 544 | 0.2005 | 2206 | 0.7927 | 26942 | 20.8172 |
| WT_LF | 554 | 0.1170 | 799 | 0.8020 | 7523 | 44.7923 |
| WT_RC | 170 | 0.0482 | 618 | 0.4587 | 2136 | 7.2476 |
| WT_SE | 174 | 0.0431 | 581 | 0.4702 | 2257 | 7.2362 |
| Σ | 28/28 | 28/28 [18] | 25/28 | 12/28 [18] | 21/28 | 12/28 [18] |

Wallace Tree (WT). Timeouts are marked with (T.O.). As shown, the only MAC configurations with higher resource requirements are the CL (FSA) for all AR, CWT, WT, and DT from 8-bit onwards and CS for 16 and 32-bit. For the 16 and 32-bit designs the DT_CL and DT_CS result in timeouts. The final row compares how many MAC designs using the proposed approach are verifiable compared to the work of [18]. For the 8-bit case, all designs are verifiable using both the proposed method and the approach in [18]. However, from 16-bit onwards, [18] is only able to verify fewer than half of the designs, whereas only three designs timed out using our proposed method. For 32-bit designs, our method successfully verifies nine more designs than [18].

Behavioral MAC designs have been especially challenging to verify in the past [19], therefore in the following Table V the proposed approach is compared to existing work. From left to right the table is comprised of the following columns. The first two describe the *bit width* (B) and *number of nodes* (NN). Then groups for substitution order and extension categorizes: index, index+phase, dynamic, dynamic+phase, dynamic+phase+conflict which are compared to extensions of RevSCA for MAC [18] and Transformation [19]. For each a subset of the following columns are presented: *Maximum Polynomial* (MP) size, *Verification Time* (VT), number of *Polarity Negations* (PN), *Dynamic Steps* (DS), *Number of Conflicts* (NC). Dynamic Steps represent the total number of tested substitutions divided by the number of substitution steps. A higher value implies that the dynamic ordering had to test different substitutions more often.

So far, the largest verifiable behavioral MAC was 11-bits, achieved using a transformation approach [19] to manage polynomial size explosion. This outperformed the extended RevSCA, which could only verify up to 9-bits. To demonstrate the importance of substitution ordering, the table presents both the index-based ordering and the proposed dynamic ordering, both performed using our gate-based SCA-based proof engine. Using the baseline index substitution, as well as the index substitution with phase extension, only 8-bit MAC designs can be verified. However, when employing dynamic ordering, up to 32-bit designs can be verified. In this case, the verification process requires only ∼19 seconds (VT) with a MP size of 8401. Applying the phase extension to this dynamic ordering also allows 32-bit designs to be verified, but increasing the MP and VT. When dynamic ordering is combined with both phase and conflict extensions, the MP size is reduced significantly further to just 2698, and the VT drops to 3.4 seconds. Furthermore, it also allows for the verification of 64-bit MAC in only 833 seconds while maintaining a reasonably small MP. This represents a tremendous improvement over both RevSCA extension and the transformation approach. For their largest verifiable MAC designs, RevSCA extension reached an MP size of 5.8 million monomials, while the transformation approach reached 50k monomials. These results highlight the effectiveness and importance of a robust dynamic substitution ordering, alongside advanced techniques to mitigate polynomial expansion through phase and conflict handling, which is a major issue in SCA-based verification.

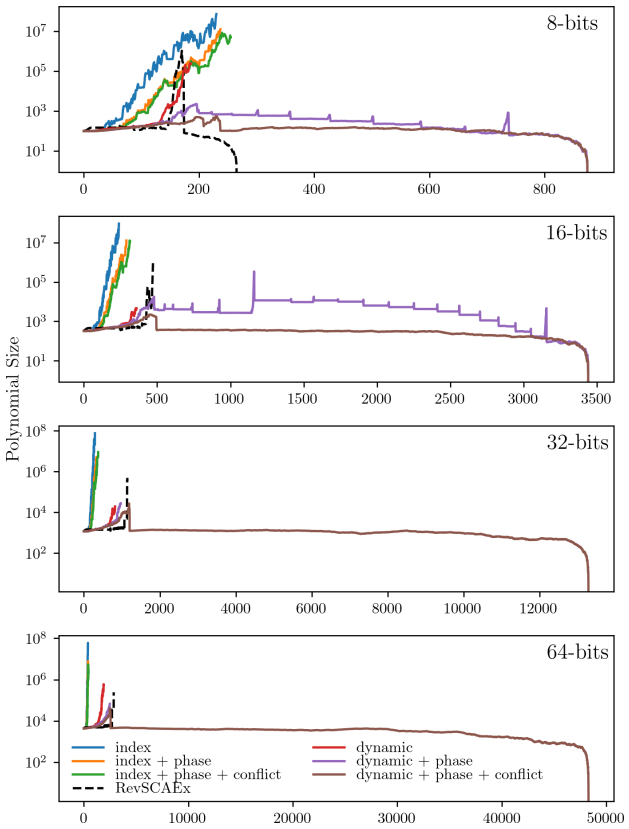



Fig. 7: Substitution Curve Example for WT_KS MAC using Index and Dynamic Substitution Ordering

To better understand why the combination of dynamic ordering and conflict extensions succeeds, where other methods fail, Fig. 7 illustrates the polynomial size evolution across substitution steps for the Wallace tree and Kogge-Stone (WT_KS) MAC using various ordering techniques. Two substitution orderings are presented: index-based (idx) and dynamic (dyn) ordering. Both are shown in three configurations: without additional techniques, with phase extension (+p), and with both phase and conflict extensions (+p+c). These results are compared to the RevSCA (extended) approach from [19]. While the RevSCA extension utilizes atomic-block substitution which leads to reduced substitution steps, our gate-based method scales with the number of gates and therefore results in larger numbers of substitution steps. From top to bottom, the figure displays MAC designs for 8, 16, 32, and 64-bit widths. The x-axis represents the substitution steps, while the y-axis indicates the number of monomials (size) in the polynomials.

For 8-bit designs, it is evident that all index-based methods grow rapidly and eventually time out. Only the dyn+p, dyn+p+c, and RevSCA extended configurations successfully verify the design. For 16-bit and 32-bit designs, the index-based methods and RevSCA extended, result in timeouts; only dyn, dyn+p and dyn+p+c achieve verification. Finally, for 64-bit designs, only dyn+p+c successfully verifies the designs

TABLE V: MAC Verification Results for Different Substitution Orderings and Optimizations

| | | Index | | Index+Phase | | | Dynamic | | | Dynamic+Phase | | | | Dynamic+Phase+Conflict | | | | | RevSCA [18] | | Transform. [19] | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B | NN | MP | VT | MP | VT | PN | MP | VT | DS | MP | VT | PN | DS | MP | VT | PN | DS | NC | MP | VT | MP | VT |
| 2 | 38 | 30 | 0.001 | 24 | 0.002 | 22 | 36 | 0.002 | 1.158 | 26 | 0.001 | 17 | 1.000 | 21 | 0.002 | 11 | 1 | 14 | 19 | 0.001 | 13 | 0.001 |
| 3 | 81 | 60 | 0.002 | 33 | 0.002 | 47 | 58 | 0.003 | 1.111 | 38 | 0.002 | 41 | 1.000 | 34 | 0.003 | 19 | 1 | 30 | 29 | 0.002 | 23 | 0.002 |
| 4 | 147 | 650 | 0.004 | 233 | 0.015 | 87 | 74 | 0.005 | 1.075 | 63 | 0.006 | 77 | 1.034 | 48 | 0.009 | 42 | 1 | 55 | 49 | 0.003 | 34 | 0.002 |
| 5 | 237 | 42777 | 0.280 | 3300 | 0.251 | 157 | 165 | 0.012 | 1.181 | 91 | 0.013 | 120 | 1.038 | 67 | 0.009 | 56 | 1 | 89 | 4297 | 0.115 | 50 | 0.006 |
| 6 | 334 | 41741 | 0.213 | 3233 | 0.139 | 205 | 192 | 0.010 | 1.009 | 145 | 0.023 | 171 | 1.018 | 82 | 0.013 | 81 | 1 | 126 | 418960 | 176.248 | 90 | 0.009 |
| 7 | 451 | 1385673 | 13.365 | 29103 | 2.790 | 289 | 327 | 0.036 | 1.100 | 236 | 0.089 | 233 | 1.126 | 108 | 0.021 | 99 | 1 | 168 | 732122 | 274.251 | 146 | 0.009 |
| 8 | 596 | 21398915 | 354.629 | 253258 | 73.157 | 401 | 393 | 0.085 | 1.190 | 373 | 0.143 | 305 | 1.104 | 138 | 0.039 | 133 | 1 | 224 | 5819510 | 1445.410 | 336 | 0.033 |
| 9 | 751 | T.O. | T.O. | T.O. | T.O. | T.O. | 364 | 0.115 | 1.153 | 350 | 0.168 | 383 | 1.108 | 155 | 0.054 | 186 | 1 | 285 | 5888912 | 2472.290 | T.O. | T.O. |
| 10 | 919 | T.O. | T.O. | T.O. | T.O. | T.O. | 434 | 0.073 | 1.062 | 586 | 0.367 | 469 | 1.099 | 181 | 0.072 | 220 | 1 | 346 | T.O. | T.O. | 1261 | 0.128 |
| 11 | 1106 | T.O. | T.O. | T.O. | T.O. | T.O. | 576 | 0.165 | 1.190 | 738 | 0.439 | 561 | 1.142 | 215 | 0.103 | 270 | 1 | 419 | T.O. | T.O. | 50908 | 15.624 |
| 12 | 1312 | T.O. | T.O. | T.O. | T.O. | T.O. | 1017 | 0.263 | 1.127 | 1047 | 0.833 | 669 | 1.125 | 278 | 0.179 | 279 | 1 | 664 | T.O. | T.O. | T.O. | T.O. |
| 16 | 2295 | T.O. | T.O. | T.O. | T.O. | T.O. | 1487 | 0.808 | 1.131 | 1908 | 2.539 | 1164 | 1.121 | 458 | 0.196 | 436 | 1 | 1015 | T.O. | T.O. | T.O. | T.O. |
| 32 | 8870 | T.O. | T.O. | T.O. | T.O. | T.O. | 8401 | 18.858 | 1.166 | 44367 | 1055.62 | 4514 | 1.507 | 2698 | 3.431 | 1830 | 1.01 | 4967 | T.O. | T.O. | T.O. | T.O. |
| 64 | 34521 | T.O. | T.O. | T.O. | T.O. | T.O. | T.O. | T.O. | T.O. | T.O. | T.O. | T.O. | T.O. | 34274 | 833.401 | 7060 | 1.06 | 367991 | T.O. | T.O. | T.O. | T.O. |

while maintaining a reasonably small maximum polynomial size. This emphasizes the effectiveness of the conflict optimization technique when combined with phase extension and dynamic ordering, particularly when compared to other techniques and RevSCA extension [19].

## VI. Conclusion

The proposed gate-based SCA-based proof engine effectively addresses the challenge of intermediate term explosion by leveraging a combination of conflict optimization, polarity and dynamic ordering for MAC designs. The novel conflict optimization learns the signal dependencies which drastically reduces the polynomial explosion. Our experimental results reveal that these optimizations do not merely provide incremental speedups but represent a critical advancement in verification capability, particularly for complex designs which were not verifiable before. Our findings emphasize the potential of advanced polynomial compression techniques particularly the identification of conflicts and its removal to enhance the verifiability.

## Acknowledgment

This work was supported in part by DFG within the Reinhart Koselleck Project PolyVer (DR 287/36-1) and partly by the Federal Ministry of Research, Technology and Space (BMFTR) within the ExaVerse project under grant no. 01IW25003 and project DI-ReDesign under grant no. 16ME0949.

## References

[1] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *DAC*, 2001, pp. 530–535.

[2] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *TC*, vol. 35, no. 8, pp. 677–691, 1986.

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

[4] ——, "Incremental column-wise verification of arithmetic circuits using computer algebra," *FM*, vol. 56, no. 1, pp. 22–54, 2020.

[5] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA-2.0: SCA-Based Formal Verification of Nontrivial Multipliers Using Reverse Engineering and Local Vanishing Removal," *TCAD*, vol. 41, no. 5, pp. 1573–1586, 2022.

[6] A. Konrad and C. Scholl, "Symbolic Computer Algebra for Multipliers Revisited-It's All About Orders and Phases," in *FMCAD*, 2024, pp. 261–271.

[7] R. Li, L. Li, H. Yu, M. Fujita, W. Jiang, and Y. Ha, "RefSCAT: Formal Verification of Logic-Optimized Multipliers via Automated Reference Multiplier Generation and SCA-SAT Synergy," *TCAD*, vol. 44, no. 2, pp. 791–804, 2024.

[8] H. Liu, P. Liao, J. Huang, H.-L. Zhen, M. Yuan, T.-Y. Ho, and B. Yu, "Parallel Gröbner Basis Rewriting and Memory Optimization for Efficient Multiplier Verification," in *DATE*, 2024, pp. 1–6.

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] L. Weingarten, K. Datta, and R. Drechsler, "Towards Polynomial Formal Verification of Neuromorphic Architectures," in *ISED*. IEEE, 2024, pp. 1–6.

[16] ——, "Late Breaking Results: Towards Efficient Formal Verification of Dot Product Architectures," in *DATE*. IEEE, 2025, pp. 1–2.

[17] J. Kleinekathöfer, L. Weingarten, K. Datta, and R. Drechsler, "Late Breaking Results: Efficient Formal Verification of Highly Optimized MAC Units," in *DATE*. IEEE, 2026, pp. 1–3.

[18] L. Weingarten, K. Datta, and R. Drechsler, "ForMAt: Formal Verification of Scalable Multiply and Accumulate Unit," in *FDL*, 2025, pp. 1–7.

[19] ——, "Transformation-Aided Verification of MAC Designs using Symbolic Computer Algebra," in *DVCon Europe*, 2025, pp. 1–7.

[20] R. Drechsler and A. Mahzoon, "Polynomial Formal Verification: Ensuring Correctness under Resource Constraints," in *ICCAD*, 2022, pp. 70:1–70:9.

[21] R. Zimmermann and D. Tran, "Optimized synthesis of sum-of-products," in *ACSSC*, vol. 1, 2003, pp. 867–872 Vol.1.

[22] Swartzlander, "Merged arithmetic," vol. 100, no. 10, pp. 946–950, 1980.

[23] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: Clean your Polynomials before Backward Rewriting to verify Million-gate Multipliers," in *ICCAD*, 2018, pp. 1–8.

[24] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *DATE*, 2020, pp. 544–549.

[25] T. Sasao, *Logic Synthesis and Optimization*. Kluwer Academic Publisher, 1993.