Formal Verification of Integer Multiplier Circuits using Algebraic Reasoning - A Survey

Daniela Kaufmann

Abstract Digital circuits are extensively used in computers and digital systems and it is of high importance to guarantee that these circuits are correct in order to prevent issues like the famous Pentium FDIV bug. Formal verification can be used to derive the correctness of a given circuit with respect to a certain specification. However arithmetic circuits, and most prominently multipliers, impose a challenge for existing verification techniques. Currently one of the most effective techniques is based on algebraic reasoning. In this approach the circuit is modeled as a set of pseudo-Boolean polynomials and the word-level specification is reduced by a Gröbner basis, which is implied by the polynomial representation of the circuit. The circuit is correct if and only if the final result is zero. Nonetheless the verification process might not be error-free. Generating and automatically checking proofs independently increases confidence in the results of automated reasoning tools. In this paper we survey the current state of the art of this work. We give an overview over recent solving techniques, available benchmarks, and include a comprehensive evaluation.

1 Introduction

Digital circuits carry out logical operations, which make them an important component in computers and digital systems, because they represent models for various digital components and arithmetic operations. The basic function of a digital circuit is to compute binary digital values for the logical function it implements, given binary values at the input. The computation is usually realized by logic gates that represent simple Boolean functions, such as negation (NOT), conjunction (AND), disjunction (OR), or exclusive disjunction (XOR). These logic gates can be combined to build more complex logical operations. A subclass of digital circuits are combinational logic circuits, where the output of the circuit is a function of the present

Daniela Kaufmann
Johannes Kepler University Linz, Austria e-mail: daniela.kaufmann@jku.at

input only, i.e., the output does not depend on previous input values. Combinational logic is used in computer circuits to perform Boolean algebra. For example, the part of an arithmetic logic unit (ALU) in a CPU, which is responsible for mathematical calculations, is constructed using combinational logic. If a circuit implements an arithmetic operation, it is called an *arithmetic circuit*, which can be further refined to determine specific arithmetic operations such as *adder circuits* or *multiplier circuits*.

Since these circuits are such a crucial part of processors, it is extremely important to guarantee their correctness in order to prevent issues like the famous Pentium FDIV bug [49] that was detected in 1994. This bug affected the floating point unit of early Intel Pentium processors. The division algorithm for floating points used a lookup table to calculate the intermediate quotients. Due to a programming error, five entries of the lookup table contained zero instead of +2. Thus the result was incorrect and in the worst case the error could affect the fourth significant digit of a decimal number. Even more than 25 years after detecting this bug, automatically proving the correctness of arithmetic circuits, and especially multiplier circuits, is still considered to be a challenge.

Formal verification can be used to prove or disprove the correctness of a given system with respect to a predefined specification. To this end the system is translated into a mathematical model and automated decision processes are applied to derive the desired correctness property. The different formal verification approaches are distinguished by the mathematical formalism used in the verification process.

Up to now several solving techniques have been developed for multiplier verification. The first technique that was shown to detect the Pentium bug is based on binary decision diagrams [10], more precisely on binary moment diagrams (BMDs) [13] and variants [14], since their size remains linear in the number of input bits of a multiplier. However, this approach requires structural knowledge of the multipliers [11, 13]. It is important to determine the order in which BMDs are built, because it has tremendous influence on the size and thus performance.

A common approach models the problem as a satisfiability (SAT) problem, where the circuit is translated into a formula in conjunctive normal form (CNF). A large set of such encodings was submitted to the SAT Competition 2016 [7]. The results indicated that verifying CNF miters of multipliers needs exponential sized resolution proofs [8], which implies exponential run-time of CDCL SAT solvers. For simple multiplier architectures, this conjecture is neglected in theory in [5], where it was shown that ring properties do admit polynomial sized resolution proofs. Recent work shows that pseudo-Boolean solvers can verify the word-level equivalence of simple multiplier architectures that consist only of half- and full-adders [32]. This method is so far not applicable for more complex architectures.

A further approach is based on the usage of theorem provers, such as ACL2 [30]. Theorem provers in combination with SAT are able to certify industrial multipliers [22]. Typically, theorem provers are not fully automated and require domain knowledge. Recently, progress has been made in the theorem prover ACL2 [52], which now allows automated verification of a large set of multiplier architectures. However, the multipliers have to be given as SVL netlists, which rely on the preservation of hierarchical information of the circuits.

Approaches based on bit-level reverse engineering [45, 50] use arithmetic bit-level representations, which are extracted from the gate-level netlists. They are able to verify simple multipliers, but fail to verify non-trivial multipliers. Methods based on term rewriting [53] require domain knowledge and thus are not fully automated.

The currently most effective technique for automated verification of flattened multipliers is based on computer algebra, e.g., [15,24,40]. In this method all gates of the circuit and its specification are represented by polynomials. If the gate polynomials are ordered according to their topological appearance, they generate a Gröbner basis [12]. Hence, the question whether a multiplier circuit is correct can be answered by reducing the specification by the implied Gröbner basis. The multiplier is correct if and only if the reduction returns zero. The main issue of the general algebraic approach is that the size of the intermediate reduction results increases drastically. Thus, several preprocessing techniques and reduction methods have been developed in recent years [15,24,40], which attempt to overcome this issue.

Nonetheless, the verification process might not be error-free. Generating and checking proofs independently increases trust in the results of automated reasoning tools. Polynomial proofs can be obtained as a by-product of verifying multiplier circuits [25,28] and can be checked by independent proof checking tools.

In this paper we survey over the current state of the art in verifying integer multipliers using computer algebra. For verification of Galois field multipliers we refer to [33,34,58,59]. In Sect. 2 we introduce the technique of circuit verification based on algebraic reasoning and present available proof formats. In Sect. 3 we present recent verification tools and discuss their strategies to overcome the issue of monomial blow-up in the intermediate reduction results. We show available benchmark generators in Sect. 4 and conclude with a comprehensive evaluation in Sect. 5.

2 Circuit verification using Computer Algebra

In this section we introduce multiplier circuits and discuss architectural details. We present the algebraic concepts that are needed in the technique of automated circuit verification using computer algebra. Furthermore, we introduce algebraic proof systems that can be used to validate the correctness of the verification results.

2.1 Multiplier Circuits

A digital circuit implements a logical function and computes binary digital values, given binary values at the input. The computation of the function is realized by logic gates, such as NOT, AND, OR, and XOR. The specification of a circuit is the desired relation between its inputs and outputs. A circuit *fulfills a specification* if for all inputs it produces outputs that match this desired relation. The goal of verification is to formally prove that the circuit fulfills its specification.

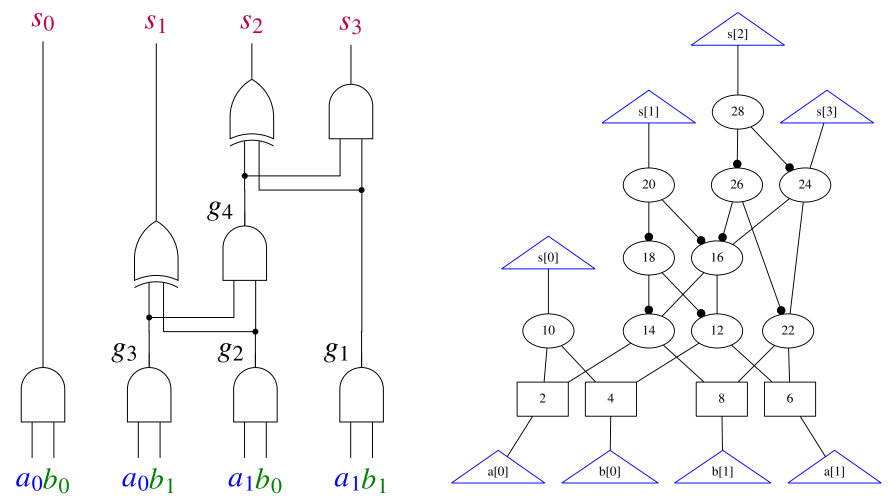


Fig. 1 Gate-level (left) and AIG (right) representation of a 2-bit multiplier circuit [24]

In this paper, we consider gate-level integer multipliers with input bits $a_0, \ldots, a_{n-1}$, $b_0, \ldots, b_{n-1} \in \{0, 1\}$ and $2n$ output bits $s_0, \ldots, s_{2n-1} \in \{0, 1\}$. If the circuit represents multiplication of unsigned integers, the multiplier is correct if and only if for all possible inputs the specification $\mathcal{U}_n = 0$ holds, where:

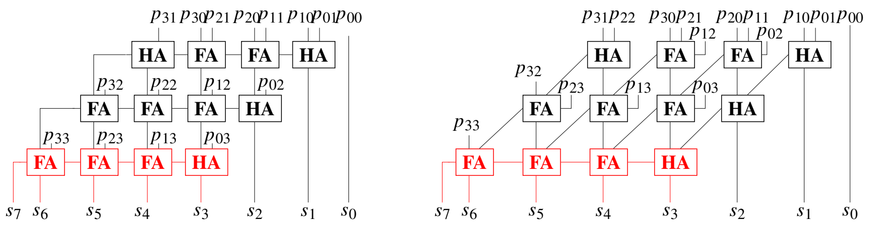
$$\mathcal{U}_n = -\sum_{i=0}^{2n-1} 2^i s_i + \left(\sum_{i=0}^{n-1} 2^i a_i\right)\left(\sum_{i=0}^{n-1} 2^i b_i\right) \tag{1}$$

Example 1 The left side of Fig. 1 shows the gate-level representation of a 2-bit unsigned integer multiplier. The variables a_1, a_0, b_1, b_0 represent the input bits of the multiplier and s_3, s_2, s_1, s_0 are the binary outputs of the multiplier. The word-level specification of this circuit is $-8s_3 - 4s_2 - 2s_1 - s_0 + (2b_1 + b_0)(2a_1 + a_0) = 0$.

If the circuit represents signed multiplication, we have to take into account that the integers in the specification $\mathcal{S}_n$ are represented using two's complement.

$$\mathcal{S}_n = -2^{2n-1} s_{2n-1} + \sum_{i=0}^{2n-2} 2^i s_i - \left(-2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i\right)\left(-2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i\right) \tag{2}$$

A common representation of combinational circuits is the encoding as an and-inverter-graph (AIG) [31]. An AIG is a directed acyclic graph, which consists of two-input nodes representing logical conjunction. The edges may contain a marking that indicates logical negation. The AIG representation usually contains more nodes, than the gate-level representation, but has an unequivocal syntax and semantics, and is very efficient to manipulate. The right side of Fig. 1 shows the AIG representation of the gate-level multiplier that is depicted on the left side.



p_{31} p_{30} p_{21} p_{20} p_{11} p_{10} p_{01} p_{00}
HA FA FA HA
p_{32} p_{22} p_{12} p_{02}
FA FA FA HA
p_{33} p_{23} p_{13} p_{03}
FA FA FA HA
s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0
p_{31} p_{22} p_{30} p_{21} p_{20} p_{11} p_{10} p_{01} p_{00}
p_{12} p_{02}
HA FA FA HA
p_{32} p_{23} p_{13} p_{03}
FA FA FA HA
p_{33}
FA FA FA HA
s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

Fig. 2 Architecture of array multipliers (left) and diagonal multipliers (right) [24]

The space and time complexity of a multiplier depends on its architecture. In general a multiplier circuit can be divided into three parts [44]. In the first component, *partial product generation* (PPG), the partial products $a_i b_j$ for $0 \leq i < n, 0 \leq j < n$, as contained in the specification, are generated. This can for example be achieved by using simple AND-gates or using a more complex Booth encoding [44].

In the second component, *partial product accumulation* (PPA), the partial products are reduced to two layers by multi-operand addition using half-adders (HA), full-adders (FA), and compressors. Well-known accumulation structures are for example array or diagonal accumulation, Wallace trees, or compressor trees [44].

In the *final-stage adder* (FSA) the output of the circuit is computed using an adder circuit. Generally, adder circuits can be split into two groups: either the carries are computed alongside the sum bits or they are calculated before the sums. Adders of the first group consist of a sequence of half- and full-adders, giving them a simple but inefficient structure. Examples are ripple-carry or carry-select adders. In order to decrease the latency of carry computation, the adder circuits of the second group precompute the carry bits of the adder. They are called *generate-and-propagate (GP) adders*. Examples are carry look-ahead adders and Kogge-Stone adders [44].

We call multipliers, that can be fully decomposed into half- and full-adders *simple multipliers*, all other architectures are called *complex multipliers*.

Example 2 We show two simple multiplier architectures with input bit-width 4 in Fig. 2. In both circuits the PPG uses AND-gates, i.e., $p_{ij} = a_i \wedge b_j$. In array multipliers, which are shown on the left side, the partial products are accumulated using a grid-like structure. The multiplier on the right side, uses a diagonal structure. In both multipliers, the FSA is a ripple-carry adder, which is highlighted in red.

2.2 Algebra

Let us now briefly summarize algebraic concepts, following [18]. Throughout this section let $\mathbb{K}[X] = \mathbb{K}[x_1, \ldots, x_n]$ denote the ring of polynomials in variables $x_1, \ldots, x_n$ with coefficients in a field $\mathbb{K}$.

Definition 1 A *term* τ is a product of the form $\tau = x_1^{e_1} \cdots x_n^{e_n}$ for $e_1, \ldots, e_n \in \mathbb{N}$. A *monomial* $m = \alpha\tau$ is a constant multiple of a term, with $\alpha \in \mathbb{K}$. A *polynomial* $p = m_1 + \cdots + m_s$ is a finite sum of monomials.

On the set of terms an order $\leq$ is fixed such that for all terms τ, σ_1, σ_2 we have $1 \leq \tau$ and $\sigma_1 \leq \sigma_2 \Rightarrow \tau\sigma_1 \leq \tau\sigma_2$. A term order is called a *lexicographic term order* if for all terms $\sigma_1 = x_1^{u_1} \cdots x_n^{u_n}$, $\sigma_2 = x_1^{v_1} \cdots x_n^{v_n}$ we have $\sigma_1 < \sigma_2$ if and only if there exists an index i with $u_j = v_j$ for all $j < i$, and $u_i < v_i$. Every polynomial $p \neq 0$ contains only finitely many terms, the largest of which (with respect to the chosen order $\leq$) is called the *leading term* and denoted by $\mathrm{lt}(p)$. If $p = \alpha\tau + \cdots$ and $\mathrm{lt}(p) = \tau$, then $\mathrm{lc}(p) = \alpha$ is called the *leading coefficient* and $\mathrm{lm}(p) = \alpha\tau$ is called the *leading monomial* of p. We call $p - \alpha\tau$ the *tail* of p.

Definition 2 A nonempty set $I \subseteq \mathbb{K}[X]$ is called an *ideal* if $\forall\, p, q \in I : p+q \in I$ and $\forall\, p \in \mathbb{K}[X]\, \forall\, q \in I : pq \in I$. If $I \subseteq \mathbb{K}[X]$ is an ideal, then a set $P = \{p_1, \ldots, p_m\} \subseteq \mathbb{K}[X]$ is called a *basis* of I if $I = \{q_1 p_1 + \cdots + q_m p_m \mid q_1, \ldots, q_m \in \mathbb{K}[X]\}$. We say *$I$ is generated by P* and write $I = \langle P \rangle$.

The theory of Gröbner bases offers a decision procedure for the so-called ideal membership problem, i.e., given $q \in \mathbb{K}[X]$ and a basis $P = \{p_1, \ldots, p_m\} \subseteq \mathbb{K}[X]$, decide whether q belongs to the ideal generated by $p_1, \ldots, p_m$. If $\{p_1, \ldots, p_m\}$ is a Gröbner basis, then the question can be answered using a multivariate version of polynomial division with remainder (cf. Thm. 3 in Chap. 2 §3 of [18]).

Definition 3 A basis $P = \{p_1, \ldots, p_m\}$ of an ideal $I \subseteq \mathbb{K}[X]$ is called a *Gröbner basis* (with respect to a fixed order $\leq$) if and only if $\forall q \in I \exists p_i \in P : \mathrm{lm}(p_i) \mid \mathrm{lm}(q)$.

Lemma 1 *Every ideal $I \subseteq \mathbb{K}[X]$ has a Gröbner basis with respect to a fixed order $\leq$.*

Proof Cor. 6 in Chap. 2 §5 of [18].

Given an arbitrary basis of an ideal, Buchberger's algorithm [12] is able to compute a Gröbner basis for it in finitely many steps.

Lemma 2 *If $P = \{p_1, \ldots, p_m\}$ is a Gröbner basis, then every $f \in \mathbb{K}[X]$ has a unique remainder r with respect to P. Furthermore it holds that $f - r \in \langle P \rangle$.*

Proof Prop. 1 in Chap. 2 §6 of [18].

Ultimately the following Lemma provides the answer on how we can solve the ideal membership problem with the help of Gröbner basis and thus can check whether a polynomial belongs to an ideal or not.

Lemma 3 *Let $P = \{p_1, \ldots, p_m\} \subseteq \mathbb{K}[X]$ be a Gröbner basis, and let $f \in \mathbb{K}[X]$. Then f is contained in the ideal $I = \langle P \rangle$ if and only if the remainder of f with respect to P is zero.*

Proof Cor. 2 in Chap. 2 §6 of [18].

2.3 Circuit Verification using Computer Algebra

In this section we introduce the technique of circuit verification using computer algebra, following [26]. We consider circuits C with inputs $a_0, \dots, a_{n-1}$ and $b_0, \dots, b_{n-1}$, outputs $s_0, \dots, s_{2n-1}$, and a number of logical gates, denoted by $g_1, \dots, g_k$. By R we denote the ring $\mathbb{K}[a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1}, g_1, \dots, g_k, s_0, \dots, s_{2n-1}] = \mathbb{K}[X]$.

The semantics of each circuit gate implies a polynomial relation among the input and output variables, such as the following ones:

$$\begin{array}{lll} u = \neg v & \text{implies} & 0 = -u + 1 - v \\ u = v \wedge w & \text{implies} & 0 = -u + vw \\ u = v \vee w & \text{implies} & 0 = -u + v + w - vw \\ u = v \oplus w & \text{implies} & 0 = -u + v + w - 2vw. \end{array} \tag{3}$$

We call these polynomials *gate polynomials* or *gate constraints*. Let $G(C) \subseteq R$ denote the set of polynomials, which contains for each gate of the given circuit the corresponding polynomial of (3).

Example 3 The possible solutions for the gate constraint $p_{00} = a_0 \wedge b_0$ represented as (p_{00}, a_0, b_0) are $(1, 1, 1), (0, 1, 0), (0, 0, 1), (0, 0, 0)$ which are all solutions of the polynomial $-p_{00} + a_0 b_0 = 0$, when a_0, b_0 are restricted to the Boolean domain.

All variables $x \in X$ are Boolean and we enforce this property by assuming the set $B(X) = \{x(1 - x) \mid x \in X\} \subseteq R$ of *Boolean value constraints*.

Since the logical gates are functional, the values of $g_1, \dots, g_k, s_0, \dots, s_{2n-1}$ in a circuit are determined as soon as the inputs $a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1} \in \{0, 1\}$ are fixed. This motivates the following definition of polynomial circuit constraints [26].

Definition 4 Let C be a circuit. A polynomial $p \in R$ is called a *polynomial circuit constraint (PCC)* for C if for every choice of

$$(a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1}) \in \{0, 1\}^{2n}$$

and the resulting values $g_1, \dots, g_k, s_0, \dots, s_{2n-1}$ which are implied by the gates of the circuit C, the substitution of all these values into the polynomial p gives zero. The set of all PCCs for C is denoted by $I(C)$.

It is easy to see that $I(C)$ is an ideal of R. Since it contains all PCCs, this ideal includes all relations that hold among the values at the different points in the circuit. The circuit fulfills a certain specification $\mathcal{L}$ if and only if the polynomial relation corresponding to the specification of the circuit is contained in the ideal $I(C)$.

Thus, checking whether a given circuit C is a correct multiplier reduces to an ideal membership test. Definition 4 does not provide any information of a basis of $I(C)$, hence Gröbner basis technology is not directly applicable. However, we can deduce at least some elements of $I(C)$ from the semantics of circuit gates.

Definition 5 Let C be a circuit and assume $G(C) \subseteq R$ be the set which contains for each gate of C the corresponding polynomial of (3).
Let $B_0(G) = B(\{a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}\})$ and $J(C) = \langle G(C) \cup B_0(G) \rangle \subset R$.

Assume that we have a verifier which checks for a given circuit C and a given specification polynomial $\mathcal{L} \in R$ whether it holds that $\mathcal{L} \in J(C)$. Because it holds that $J(C) = I(C)$ [26], such a verifier is sound and complete.

Theorem 1 *Let C be a circuit, and let $J(C)$ be as in Def. 5. Furthermore let $\leq$ be a reverse topological lexicographic term order where the variables are ordered such that the variable of a gate output is always greater than the variables attached to the input edges of that gate. Then $G(C) \cup B_0(G)$ is a Gröbner basis for $J(C)$ with respect to the ordering $\leq$.*

Proof This theorem is shown for instance in [26, 34, 54]. □

Hence $G(C) \cup B_0(G)$ is a Gröbner basis for the ideal $J(C)$ and we can decide membership using Gröbner bases theory, i.e., we reduce the specification $\mathcal{L}$ by elements of $G(C) \cup B_0(G)$ until no further reduction is possible. The circuit is correct if and only if the final remainder is zero.

In this section we restricted the theory to polynomial rings over a field $\mathbb{K}$. A generalization for polynomial rings over *principal ideal domains* (such as $\mathbb{Z}$) can be found in [27], where it is furthermore discussed how to invoke modular reasoning, i.e., reasoning in rings $R = \mathbb{Z}_l[X]$. Modular reasoning allows to eliminate monomials that have large coefficients.

As a final remark, in the case when a polynomial g is not contained in an ideal $I = \langle P \rangle$, i.e., the remainder of dividing g by P is not zero and allows to determine a concrete choice of input assignments for which g does not vanish. In our application of multiplier verification these evaluations provide counter-examples, in case a circuit is determined not to be a multiplier.

2.4 Algebraic Proof Systems

Although the verification method is sound and complete, it may happen that the implementation contains errors and the reasoning engine delivers wrong results. One way to overcome this issue is to verify the implementation, e.g. [52], which is typically very tedious and requires a lot of effort. Thus, it is common to produce proof certificates in the reasoning engine to monitor the verification process. These proofs are generated as by-product of the reasoning technique and are given to independent (and ideally verified) proof checkers to validate the verification result.

For computer algebra two algebraic proof systems are used in practice, the *practical algebraic calculus* (PAC) [46] that is based on the *polynomial calculus* (PC) [17], and the *Nullstellensatz proof* format (NSS) [4].

Practical Algebraic Calculus The practical algebraic calculus [46] is an instantiated version of PC [17] which allows efficient proof checking. A PAC proof consists of three components (i) the given set of polynomials G, i.e., the constraint set (ii) the core proof, i.e., a sequence of proof rules P that model the properties of an ideal, and (iii) the target polynomial f. In a correct proof it is derived whether the target polynomial can be derived from the constraint set using the proof rules.

Initially the proof format has been defined for polynomial rings $\mathbb{K}[X]$, where $\mathbb{K}$ is a field [17, 46] and all variables represent Boolean values.

The soundness and completeness arguments have been generalized to rings $R[X]$, where all polynomials in the constraint set have unique leading terms that contain only a single variable, cf. Thm. 1 and Thm. 2 in [27]. Recently, the PAC format has been revised to derive a more compact proof representation [28].

Let P be a sequence of polynomials that can be accessed via indices. We write $P(i) = \bot$ to denote that the sequence P at index i does not contain a polynomial, and $P(i \mapsto p)$ to determine that P at index i is set to p. The initial state is $(X = \text{Var}(G \cup \{f\}), P)$ where P maps indices to polynomials of G.

[ADD (i, j, k, p)] $\quad (X, P) \implies (X, P(i \mapsto p))$

where $P(j) \neq \bot, P(k) \neq \bot, P(i) = \bot, p \in R[X]/\langle B(X)\rangle$, and $p = P(j)+P(k)$.

[MULT (i, j, q, p)] $\quad (X, P) \implies (X, P(i \mapsto p))$

where $P(j) \neq \bot$, $P(i) = \bot$, $p, q \in R[X]/\langle B(X)\rangle$, and $p = q \cdot P(j)$.

PAC proofs that are defined over $\mathbb{Z}[X]$ can be checked by the checkers PACHECK (implemented in C) [28] and PASTÈQUE (verified in Isabelle/HOL) [28].

Nullstellensatz The Nullstellensatz proof system [4] allows to derive whether a target polynomial $f \in R[X]$ can be represented as a linear combination from a given set of polynomials $G = \{g_1, \ldots, g_l\} \subseteq R[X]$ and the Boolean value constraints $B(X)$. Similar to PAC, the NSS proof system is initially defined for polynomial rings over fields [4]. By the same arguments given for PAC the soundness and completeness arguments can be generalized for rings $R[X]$ where all polynomials in G have unique leading terms that contain only one variable [25].

Again, we handle the Boolean value constraints implicitly and derive the following proof format. For a polynomial $f \in R[X]/\langle B(X)\rangle$ and a given set of polynomials $G = \{g_1, \ldots, g_l\} \subseteq R[X]/\langle B(X)\rangle$ a NSS proof is an equality P, such that

$$\sum_{i=1}^{l} h_i g_i = f \in R[X]/\langle B(X)\rangle, \tag{4}$$

with $h_i \in R[X]/\langle B(X)\rangle$.

Nullstellensatz proofs over $\mathbb{Z}[X]$ can be checked using NUSS-CHECKER [25].

3 Verification Tools

In this section we present reasoning tools for verification of flattened gate-level integer multipliers using computer algebra. We focus on the most recent work that has been developed in the last three years and consider the tools from *Yu et al.*: ABC/ARTi [15, 60]; *Kaufmann et al.*: AMulet [27]; and *Mahzoon et al.*: PolyCleaner [38], RevSCA/RevSCA-2.0 [40], DyPoSub [42] (sorted chronologically).

We discuss the scope of application of these tools and present their techniques that help to overcome the issue of monomial blow-up during reduction. All these tools are considered in the experimental evaluation in Sect. 5.

3.1 Algebraic RewriTing in ABC [15, 16, 57, 60]

The authors of [16, 57] use a method called *function extraction* to verify circuits. Function extraction is a similar algebraic approach to Gröbner basis reduction as presented in Sect. 2. The difference to Gröbner basis reduction is that it is not required to provide the complete specification polynomial of the circuit for reduction. Instead the word-level output of the circuit, i.e., the bit-vector $\sum_{i=0}^{2n-1} 2^i s_i$ for unsigned numbers resp. $-2^{2n-1} s_{2n-1} + \sum_{i=0}^{2n-2} 2^i s_i$ for signed number representation is reduced by the gate constraints of the given circuit. The Boolean value constraints are reduced implicitly, i.e., every exponent greater than one is immediately reduced to one. This method returns a unique polynomial representation of the functionality of the circuit in terms of the circuit inputs. In order to verify correctness of a circuit, this remainder polynomial needs to be compared to the desired circuit functionality.

In follow-up work [15,60] the authors introduced an optimization, where half- and full-adders are extracted by identifying subcircuits in the given circuit that represent MAJ3 and XOR3 gates. These XOR3 and MAJ3 gates are essential components of adder trees that are present in most arithmetic circuits. The polynomial constraints of all circuit gates that belong to a MAJ3 or XOR3 gate are replaced by a single polynomial that encodes a MAJ3 or XOR3 gate in order to simplify the polynomial representation of the circuit. These polynomials are sorted topologically pairwise.

The authors developed a framework called ARTi (Algebraic RewriTing) [56] that is integrated within the ABC tool [6]. Verification of multipliers that are given as AIGs is executed using the command `&polyn`, which can be configured to define whether signed or unsigned multiplication is considered. The extraction of the adder trees is invoked by the command `&atree` [15, 60].

This technique is able to handle very large multipliers that can be fully decomposed into half- and full-adders (for instance the array and diagonal multipliers of Fig. 2) efficiently, but fails on slightly optimized multiplier architectures, because invoking the command `&atree` on these multipliers leads to incompleteness. In the experimental evaluation we use `&atree` only for those benchmarks where we know that the circuit can be represented by adder trees, i.e., the experiments of Sect. 5.5.

3.2 AMulet [27]

In [27] it is presented how the verification approach can be generalized to polynomial rings that include modular reasoning, i.e., $\mathbb{Z}_l[X]$ for $l = 2^{2n}$. This allows to cancel monomials in the intermediate results with coefficients that are multiples of 2^{2n}.

The technique of [27] uses an incremental verification algorithm for circuit verification. In this method the multiplier circuit is divided into column-wise slices and the specification polynomial is split into multiple polynomials. The correctness of the circuit is shown by incrementally verifying the correctness of each slice. The main advantage of this approach is that only one small part of the global specification is used for reduction, which helps to reduce the size of the intermediate results.

Furthermore, variable elimination is applied before reduction, i.e., after assigning the gates to slices all variables that occur in only one other polynomial within the same slice are eliminated. Structures that implement a Booth encoding are detected by pattern matching and their internal gates are eliminated too.

After variable elimination the column-wise specifications are reduced by the rewritten Gröbner basis until completion. However, certain parts of the multiplier, more precisely particular final stage adders, are hard to verify using computer algebra. These adders usually contain sequences of OR-gates, which lead to an exponential blow-up of the intermediate reduction results. On the other hand, equivalence checking of adders is easy for SAT [29].

We will take a quick excursion and introduce the SAT problem following [19]:

- A *literal* l is either a positive Boolean variable x or its negation $\overline{x}$.
- A *clause* C is a finite disjunction of literals.
- A *formula in conjunctive normal form* (CNF) F is a finite conjunction of clauses.
- An *assignment* τ is a function that consistently maps the literals of F to $v \in \{\mathbf{t}, \mathbf{f}\}$, such that $\tau(x) = v \Leftrightarrow \tau(\overline{x}) = \neg v$, where $\neg\mathbf{t} = \mathbf{f}$ and $\neg\mathbf{f} = \mathbf{t}$.
- A formula evaluates to **t** if and only if every clause in the formula evaluates to **t**. A clause C evaluates to **t** if $\exists l \in C$ with $\tau(l) = \mathbf{t}$. Given a CNF formula F, the SAT problem is to decide if there exists an assignment such that F evaluates to **t**. If such an assignment exists, the formula is *satisfiable*, otherwise it is *unsatisfiable*.

Based on the observation the technique of [27] combines SAT and computer algebra. It is detected whether a multiplier contains a complex final stage adder, which is then replaced by a simple ripple-carry adder. A bit-level miter, which is expected to be unsatisfiable, is produced to verify the correctness of the replacement using SAT solvers and the rewritten multiplier is verified by computer algebra techniques.

The authors implemented a tool called AMulet [27] that is able to handle signed and unsigned multipliers given as AIGs. The tool automatically applies adder substitution and verifies the (rewritten) multiplier using computer algebra. Furthermore, AMulet is so far the only tool that is able to produce proof certificates in the PAC and NSS proof formats, cf. Sect. 2.4. In the experiments of this paper we will use the maintained version AMulet 1.5 that is currently available on GitHub [23].

3.3 PolyCleaner [38]

The work of [38] provides an extensive analysis why the number of monomials in the reduction results increases drastically, when no preprocessing is applied. The reason of the size explosion are certain monomials, called vanishing monomials, that reduce to zero later in the reduction process. The authors observe that the vanishing monomials origin from gates where the sum and carry output of a half-adder converge and the vanishing monomials remain in the intermediate reduction results until all internal gate polynomials of the half-adders have been substituted.

The work of [38] proposes a method where the converging gates are identified and the vanishing monomials are locally removed before the specification is reduced. First, for each occurring half-adder in the circuit possible converging gates are identified and the belonging input cones are determined. A polynomial is extracted for each converging gate by substituting the gate polynomials of the associated cone. Since the extracted polynomial contains the product of the sum and carry output of the corresponding half-adder, it contains the vanishing monomials. After locally removing these vanishing monomials, the specification polynomial is reduced by these vanishing-free polynomials to verify the circuit.

This method is implemented in the tool PolyCleaner [37] that is able to verify unsigned multipliers that are given as flattened Verilog modules.

3.4 RevSCA/RevSCA-2.0 [40]

The paper [40] is a follow-up work on [38]. The authors elaborate on the disadvantages of the proposed method of [38]. First, the method of [38] highly depends on the detection of the half-adders that are considered to be implemented as pairs of XOR and AND gates. The second disadvantage is that the search space for finding the converging gates is very large. Consequently, the method of [38] only works for those multipliers where all half-adders can be detected and fails for more complex multiplier circuits.

In [40] the authors generalize the detection of converging gates and propose a technique that identifies so-called atomic blocks of the multiplier, i.e. half-adders, full-adders and compressors, using reverse engineering. The detection of converging gates becomes more independent of the actual design of the atomic blocks and the search space to identify these gates can be limited. Furthermore, since not only half-adders are considered as atomic blocks, vanishing-free polynomials are not only generated for converging gate cones, but also for the outputs of atomic blocks and lead to a more compact polynomial representation.

The authors implemented a tool, called RevSCA [39] that verifies unsigned multipliers given as AIGs. The implementation has been improved and additionally supports verification of signed multipliers is supported in RevSCA-2.0 [39].

3.5 DyPoSub [42]

In contrast to the already described methods, the technique of [42], a follow up work of [40], explicitly tries to tackle the problem of verifying multiplier circuits, where logic synthesis and technology mapping is applied.

The problem with these optimized multipliers is that the clear boundaries of certain sub-structures, such as internal half- and full-adders, may be blurred. Consequently, the compact representation of these internal substructures are no longer available. For example, the discussed methods of Sect. 3.1 and Sect. 3.2 heavily rely on these boundaries, either during rewriting or defining the substitution order.

The method described in [42] tries to overcome this issue by using a *dynamic substitution order* that allows to keep the size of the intermediate reduction results on a moderate level. Before reduction is applied, the circuit gates are preprocessed as described in [40], where atomic blocks and converging gate cones are identified and vanishing-free polynomials are extracted for these cones and atomic blocks.

After preprocessing, a dynamic backward rewriting approach is applied to verify the circuit. The core idea is that for each substitution step a number of candidate polynomials is available, which maintain the overall topological sorting and guarantee that the polynomials of atomic blocks are substituted consecutively. This ensures that the gate polynomials only need to be considered once during reduction.

At each backward rewriting step, there may be several such possible candidates available. The dynamic backward rewriting chooses the reduction candidate by the number of the occurrences of the leading term in the intermediate reduction result in ascending order. After each substitution step the increase in the number of monomials is checked. If the number of monomials grows by more than 10%, the step is undone and the specification is reduced by the next candidate polynomial in line. If there is no reduction of any candidate satisfying the threshold limit, the threshold limit is increased and the process is repeated from the first candidate.

This approach is implemented in the tool DyPoSub [41] that verifies unsigned multipliers given as AIGs. The experimental data of [42] shows that this technique is able to verify industrial benchmarks created by Synopsis. Unfortunately, these benchmarks are not open-source and are not considered in Sect. 5.

4 Benchmark Generators

The evaluation of tools heavily relies on reproducible experiments and thus on publicly available benchmarks that allow to evaluate and compare the performance of the verification tools. In this section we present the currently most common used benchmarks suites, discuss how these benchmarks are processed to AIGs, cf. Sect. 4.6 and how they can be optimized using technology mapping in Sect. 4.7. An experimental evaluation of these benchmarks will be given in Sect. 5.

4.1 Arithmetic Module Generator

To the best of our knowledge, the web-based tool Arithmetic Module Generator [20, 21] offers the most comprehensive benchmark suite. The following components can be combined to gain 192 different multipliers, given as Verilog modules:

Part. Product Gen.	Part. Product Accum.	Final-Stage adder
Booth radix-4	Array	Block carry look-ahead
Simple (AND gates)	Balanced delay tree	Brent-Kung
	Dadda tree	Carry look-ahead
	Overturned-stairs tree	Carry select
	Red. binary addition tree	Carry-skip fix size
	Wallace tree	Carry-skip variable size
	(4;2) compressor tree	Conditional sum
	(7,3) counter tree	Han-Carlson
		Kogge-Stone
		Ladner-Fischer
		Ripple carry
		Ripple-block carry look-ahead

All architectures can be generated for unsigned and signed number representation, thus yielding a total of 384 multiplier architectures. The multipliers can be generated up to input bit-width 64. In addition to integer multipliers, the Arithmetic Module Generator also allows the generation of two- and multioperand adders and multiply accumulators as well as Mastrovito and Massey-Omura parallel multipliers.

4.2 GenMul

The tool GenMul is either available as a web-based [35], or as a stand-alone tool [36] and allows to generate unsigned and signed multiplier circuits in Verilog. The following components can be combined to gain 24 architectures:

Part. Product Gen.	Part. Product Accum.	Final-Stage adder
Simple (AND gates)	Array	Brent-Kung
	Counter-based Wallace tree	Carry look-ahead
	Dadda tree	Carry-skip
	Wallace tree	Kogge-Stone
		Ladner-Fischer
		Ripple carry

For signed multiplication only 23 architectures are available, as the combination "Simple – Array – Carry-skip" yields an error message. The input bit-width can be set arbitrarily large, however for input bit-widths that are larger than 128, the generation process may run for several minutes.

4.3 MultGen

MultGen is a stand-alone tool [51] and the following components can be combined to gain 24 architectures in Verilog:

Part. Product Gen.	Part. Product Accum.	Final-Stage adder
Booth radix-2	Dadda tree	Han-Carlson
Booth radix-4	Wallace tree	Kogge-Stone
Simple (AND gates)		Ladner-Fischer
		Ripple carry

All multipliers can be generated for signed and unsigned multiplication and the input bit-width can be selected arbitrarily large. In addition, MultGen is not only able to generate stand-alone multipliers, but is also able to merge four smaller stand-alone multipliers to receive a bigger multiplier. Furthermore, MultGen provides access to a fused multiply-add operation and to a generator for the dot product.

4.4 EPFL Combinational Benchmark Suite

The EPFL Combinational Benchmark Suite [1–3] has been introduced with the aim of defining a comparative standard for logic optimization and synthesis. The benchmark suite encomparates 23 combinational circuits and is divided into arithmetic, random/control and MtM (“More than ten Million gates”) parts. The arithmetic benchmarks cover a variety of arithmetic operations, such as square-root computation, division and multiplication. The arithmetic benchmarks come in different bit-widths to provide diversity in the implementation complexity, the contained multiplier circuit has an input bit-width of 64. Each circuit is distributed in Verilog, VHDL, BLIF and AIG formats.

4.5 ABC/Boolector

Both tools ABC [6] and Boolector [43] provide access to a simple array multiplier that leads to structural equivalent AIGs. The only difference is that the inputs are sorted in sequence in ABC, whereas they are sorted interleaved in Boolector.

A multiplier is generated in ABC using the command `gen` with option `-m`, which receives the input bit-width n as parameter `-N`. The command `gen` furthermore allows the generation of a ripple-carry adder, sorter, mesh or a random single-output function. The BLIF format is converted into an AIG by structural hashing (using the command `strash` of ABC).

```
abc -c "gen -N $n -m abc${n}.blif"
abc -c "read abc${n}.blif" -c strash -c "write abc${n}.aig"
```

If we generate multipliers using the SMT-solver Boolector, we have to provide an SMT encoding of the multiplier circuit (with input bit-width n). The SMT encoding is then processed by Boolector to generate the AIG circuit:

```
m=‘expr 2 \* $n‘
btor=btor${n}.btor
echo "1 var $n a" >> $btor
echo "2 var $n b" >> $btor
echo "3 uext $m 1 $n" >> $btor
echo "4 uext $m 2 $n" >> $btor
echo "5 mul $m 3 4" >> $btor
id=6
i=0
while [ $i -lt $m ]
do
  slice=$id
  echo "$id slice 1 5 $i $i" >> $btor
  id=‘expr $id + 1‘
  echo "$id root 1 $slice" >> $btor
  id=‘expr $id + 2‘
  i=‘expr $i + 1‘
done

boolector $btor -rwl=1 -dai > btor${n}.aig
```

4.6 Processing Verilog benchmarks

The available benchmark generators MultGen, GenMul and the Arithmetic Module Generator output circuits in Verilog. Most of the verification tools rely on an AIG as input format, because AIGs are easy to handle and have an unequivocal syntax and semantics. Thus, after generating the multipliers, the Verilog format needs to be processed to gain the AIG representation. We use the Yosys Open SYnthesis Suite [55] to convert the circuits from Verilog to BLIF. The command, which can be seen below, has been thankfully explained to us by Mathias Soeken.

First, the design hierarchy is checked, which is then flattened and a generic technology mapper is applied, to replace cells in the design. The output is printed in the BLIF format, which is translated to an AIG using structural hashing in ABC:

```
yosys -p "hierarchy -auto-top -check"
      -p flatten -p techmap -o <output.blif> <input.v>

abc -c "read <output.blif>" -c strash -c "write <output.aig>"
```

4.7 Optimizing Benchmarks

The multipliers that are generated using the tools presented in Sect. 4.1–4.5 follow a general structure, where the boundaries of internal structures, such as half- and full-adders can be identified. In industrial benchmarks, these circuits are optimized to reduce the number of gates and to reduce the delay of the circuits. As already mentioned in Sect. 3.5, these benchmarks are often not publicly available.

As a compromise, we can optimize the generated circuits, e.g., in ABC, and apply rewriting and technology mapping. Optimizing the benchmarks has been thankfully explained to us by Maciej Ciesielski.

Rewriting without technology mapping is applied as follows:

```
abc -c "read <input.aig>"
    -c <syn>
    -c "write <output.aig>"
```

where `<syn>` is for instance replaced by `dc2`, `resyn`, `resyn2`, or `resyn3`, or even a combination of them.

The option `dc2` applies combinational AIG optimization. The options `resyn`, `resyn2`, and `resyn3` are standard scripts that are included in the file "abc.rc" of the source code of ABC. In these scripts multiple rounds of technology-independent rewriting, refactoring and restructuring of the AIG are performed.

If technology mapping should be involved, a standard cell library needs to be provided, e.g., "mcnc.genlib" of SIS [48] that is available at [47]. The cell library is parsed and technology mapping can be applied by the command `map`. Structural hashing is applied before the output is printed:

```
abc -c "read <input.aig>"
    -c <syn>
    -c "read <genlib>"
    -c "map"
    -c strash
    -c "write <output.aig>"
```

5 Evaluation

In this chapter we evaluate the presented tools ABC, AMULET 1.5, DYPOSUB, POLYCLEANER, REVSCA/REVSCA-2.0 of Sect. 3 using the benchmarks we have presented in Sect. 4. In our experiments we use an Intel Xeon E5-2620 v4 CPU at 2.10 GHz (turbo-mode disabled) with a memory limit of 128 GB.

We measure the time from starting a tool until it is finished and the time is listed in rounded seconds (wall-clock time). For a run of AMULET 1.5, where several tool applications are used in the verification flow, we measure the time AMULET needs to apply adder substitution and circuit verification and include the time the SAT solver KISSAT [9] needs to verify the equivalence of the adders.

5.1 Arithmetic Module Generator

In our first experiment we consider the 384 unsigned and signed 64-bit multipliers that are generated using Arithmetic Module Generator [20]. We process the Verilog format to AIG as discussed in Sect. 4.7. The tool PolyCleaner reads flattened Verilog moudules. We flatten the circuit in Yosys, using the presented command in Sect. 4.7. We write the design to a Verilog file by exchanging the suffix "`.blif`" to "`.v`", i.e. we write the circuit to `<output.v>`.

We set the time limit in the experiments to 300 seconds. The results can be seen in the CDF-plots that are shown in Fig. 3.

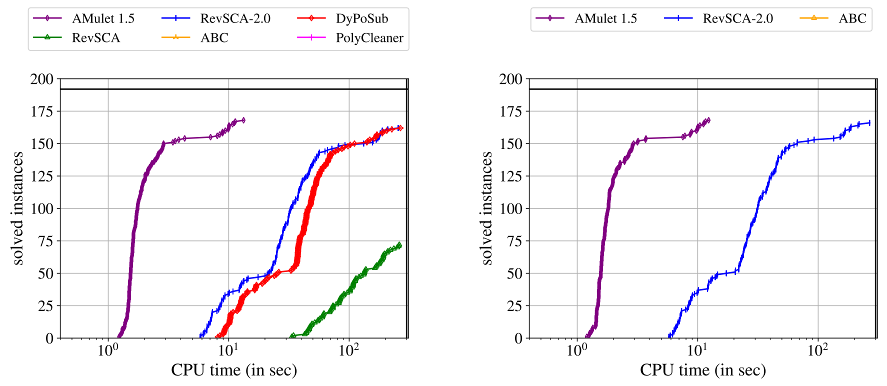


Fig. 3 Verification time (in sec) of unsigned (left) and signed (right) multipliers that are generated using the Arithmetic Module Generator. Time limit: 300 seconds

All presented tools can be applied for verification of unsigned multipliers. However, only ABC, AMulet 1.5, and RevSCA-2.0 support verification of signed multipliers. These are the only tools used in the right part of Fig. 3.

It can be seen that AMulet 1.5, DyPoSub, and RevSCA-2.0 solve the most benchmarks. More precisely, AMulet 1.5 solves 170 unsigned instances, DyPoSub and RevSCA-2.0 solve 165 unsigned benchmarks. In both experiments AMulet 1.5 is the fastest tool by an order of magnitude.

As discussed in Sect. 3.1, we do not apply the optimization `&atree`, which has the effect that ABC produces an internal error or a segmentation fault for all benchmarks. Thus there are no results for ABC. The tool PolyCleaner exceeds the time limit in all experiments in the left side of Fig. 3.

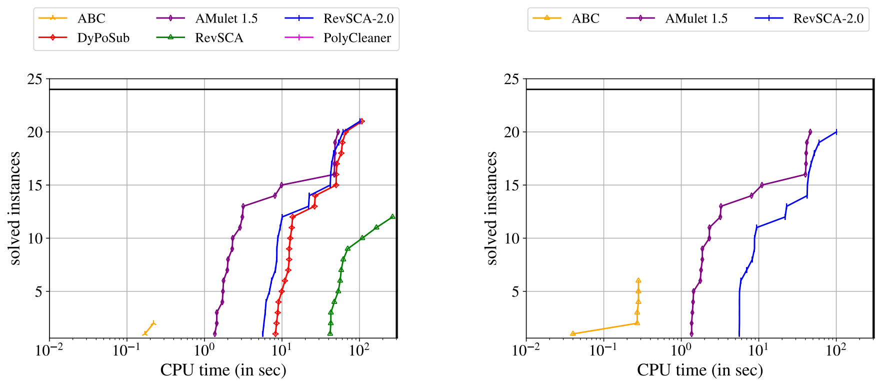


Fig. 4 Verification time (in sec) of unsigned (left) and signed (right) multipliers that are generated using GenMul. Time limit: 300 seconds

5.2 GenMul

In our second experiment we consider the 24 unsigned and 23 signed benchmarks of GenMul. We generated the benchmarks for an input bit-width of 64 and set the time limit to 300 seconds. The results are depicted in Fig. 4.

In the left side of Fig. 4 it can be seen that DyPoSub and RevSCA-2.0 both solve 21 benchmarks, with RevSCA-2.0 being slightly faster. AMulet 1.5 solves 20 instances. ABC is an order of magnitude faster, but is only able to solve two instances. Again, PolyCleaner exceeds the time limit in all experiments.

For the signed multipliers ABC is again an order of magnitude faster, but only verifies 6 benchmarks. AMulet 1.5 and RevSCA-2.0 both solve 20 instances.

5.3 MultGen

In this experiment we consider the 24 unsigned and 24 signed benchmarks of MultGen. We generated the benchmarks for an input bit-width of 64 and set the time limit to 300 seconds. The results are shown in Fig. 5.

The left side of Fig. 5 shows that AMulet 1.5, DyPoSub, and RevSCA-2.0 are able to verify the complete benchmark set, with AMulet 1.5 being the fastest tool. Furthermore, AMulet 1.5 is able to verify all signed multipliers, cf. right side of Fig. 5. ABC and PolyCleaner produce a segmentation fault or exceed the time limit for all instances.

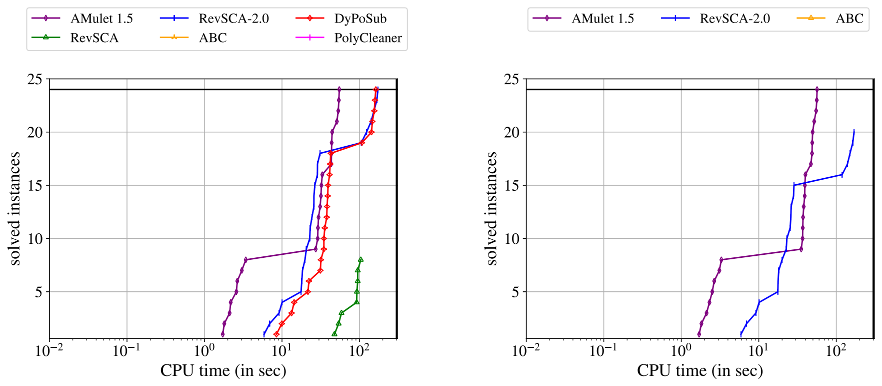


Fig. 5 Verification time (in sec) of unsigned (left) and signed (right) multipliers that are generated using MultGen. Time limit: 300 seconds

5.4 EPFL Combinational Benchmark Suite

In this experiment we verify the 64-bit multiplier that is contained in the EPFL Combinational Benchmark Suite. The time limit is set to 300 seconds and the results are shown in Tbl. 1.

Tool	ABC	AMulet 1.5	DyPoSub	PolyCleaner	RevSCA	RevSCA-2.0
CPU time (in sec)	segfault	TO	59	TO	segfault	81

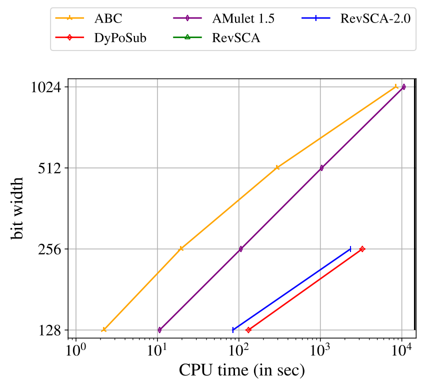
Table 1 Verification time (in sec) of the multiplier contained in the EPFL Combinational Benchmark Suite. Time limit (TO): 300 seconds

It can be seen that only DyPoSub and RevSCA-2.0 are able to verify this multiplier circuit. AMulet 1.5 and PolyCleaner exceed the time limit, ABC and RevSCA produce a segmentation fault.

5.5 ABC/Boolector

We generate simple array multipliers with an input bit-width of 128, 256, 512, and 1 024 using ABC and Boolector. Since both tools directly produce multipliers in the AIG format, we do not consider the tool PolyCleaner in this experiment. We set the time limit to 14 400 seconds (4 hours) and the results can be seen in Fig. 6.

The multipliers generated by ABC/Boolector can be fully decomposed into half- and full-adders. Thus, we enable the optimization “&atree” in the verification tool ABC. As previously mentioned, the benchmarks generated by ABC and Boolector



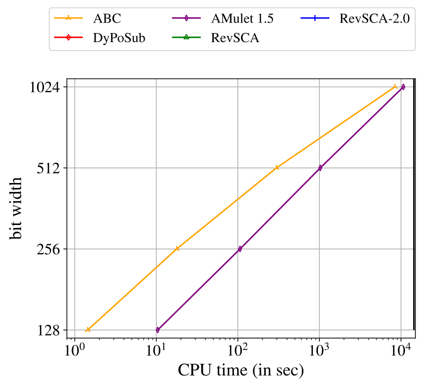


Fig. 6 Verification time (in sec) of simple array multipliers that are generated using ABC (left) and Boolector (right). Time limit: 14 400 seconds

are internally equivalent, except the order of the inputs differs. In ABC the order of inputs is $a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1}$. In Boolector the order is $a_0, b_0, \dots, a_{n-1}, b_{n-1}$.

It can be seen that ABC and AMulet 1.5 are able to verify all benchmarks, and ABC is slightly faster. The tools DyPoSub and RevSCA-2.0 are only able to verify the ABC benchmarks, but produce wrong results, i.e., conclude that the multiplier is buggy, for multipliers generated by Boolector. RevSCA produces a segmentation fault for all experiments.

5.6 Optimized Benchmarks

In this experiment we consider the 192 unsigned multiplier circuits that are generated by the Arithmetic Module Generator. The time limit is set to 300 seconds. We optimize these benchmarks as described in Sect. 4.7 and either apply “resyn”, “resyn3”, or “dc2” in ABC. In all of these benchmark suites we furthermore apply technology mapping using the standard cell library “mcnc.genlib” of SIS [48]. Thus, we gain six different set-ups and the results can be seen in Fig. 7–9.

It can be seen that DyPoSub and RevSCA-2.0 are able to verify more than half of the benchmarks when only synthesis is applied. If additionally technology mapping is used to optimize the multipliers, these tools are only able to verify at most eight circuits. The tools AMulet 1.5 and RevSCA are able to verify a small amount (less than 10) of synthesized multipliers, but fail when technology mapping is applied. ABC produces a segmentation fault or an internal error for all experiments.

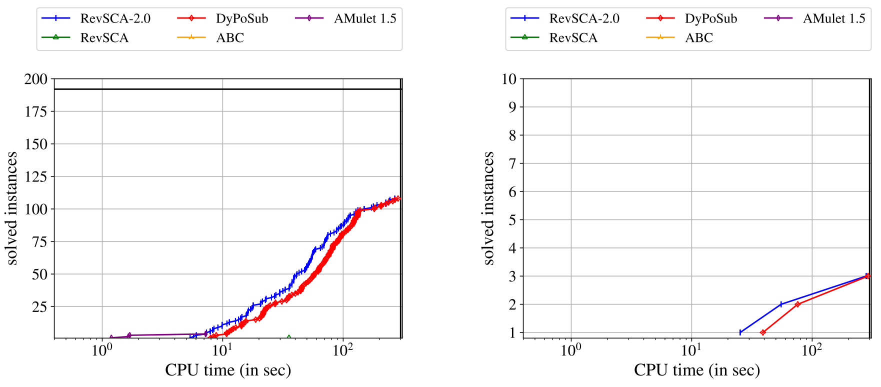


Fig. 7 Verification time of unsigned multipliers which are optimized using “resyn”, without (left) and with technology mapping (right). Time limit: 300 seconds

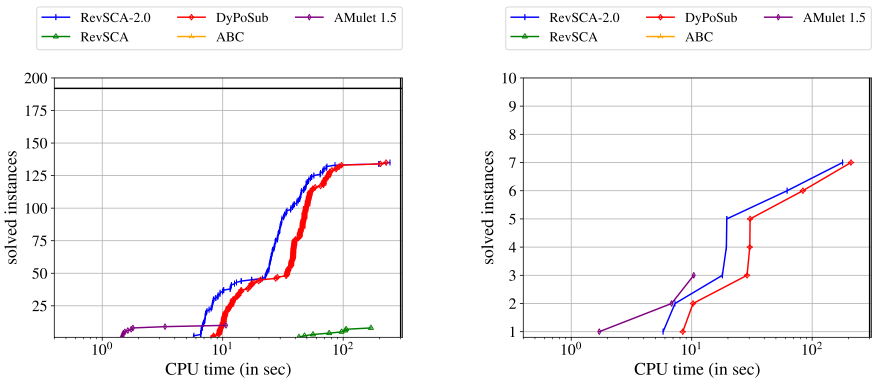


Fig. 8 Verification time of unsigned multipliers which are optimized using “resyn3”, without (left) and with technology mapping (right). Time limit: 300 seconds

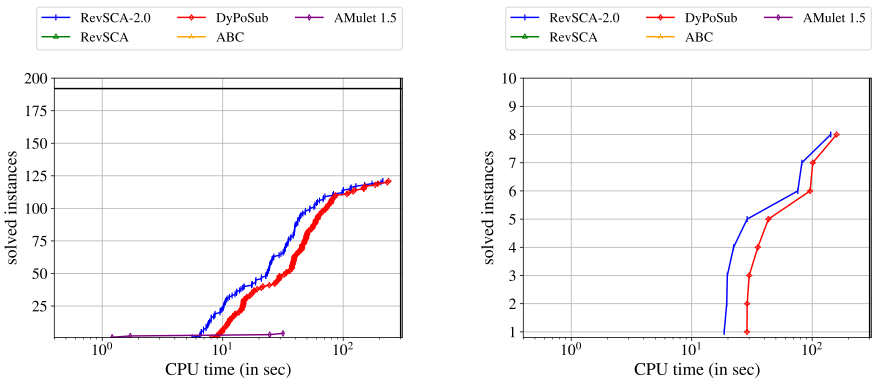


Fig. 9 Verification time of unsigned multipliers which are optimized using “dc2”, without (left) and with technology mapping (right). Time limit: 300 seconds

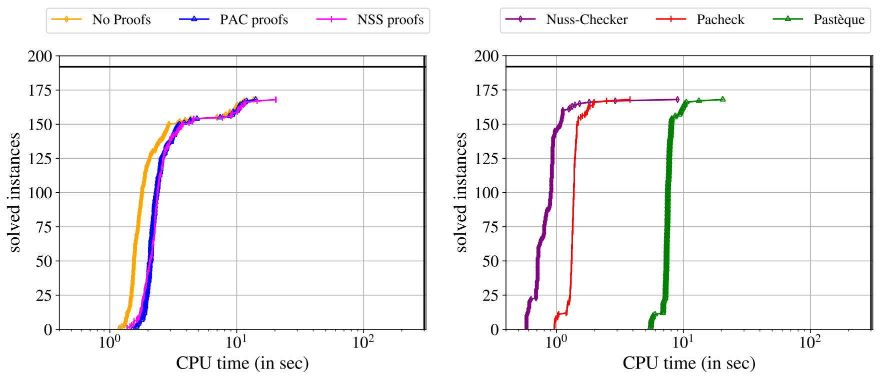


Fig. 10 Certification time of unsigned multipliers (left) and the proof checking time (right). Time limit: 300 seconds

5.7 Proof Generation

In our last experiment we present the ability of generating proofs in AMULET 1.5. Again, we consider the 192 unsigned benchmarks of the ARITHMETIC MODULE GENERATOR and produce either PAC proofs or NSS proofs, which are then checked using PACHECK [28], PASTÈQUE [28] and NUSS-CHECKER [25]. The results are shown in Fig. 10 and it can be seen that generating proof certificates increases the computation time of AMULET 1.5. Both proof formats need the same amount of time for generation. In the right side of Fig. 10 we compare the checking time of the generated proofs. Proofs in the NSS format, which are checked by NUSS-CHECKER are faster to check than PAC proofs that can be checked either by PACHECK or by PASTÈQUE. The verified proof checker PASTÈQUE is around four times slower than PACHECK.

6 Conclusion

In this paper we presented the current state of the art in verifying flattened gate-level integer multipliers using computer algebra. We introduced the verification technique and discussed the most recent work in this area. Tools that have been developed within the last three years were highlighted and publicly available benchmarks were presented. We concluded with a rigorous evaluation of the tools. Summarizing, no verification tool is a clear favorite over another. For simple multipliers, ABC [15,60] is the fastest tool, however ABC fails on complex multipliers. For non-optimized benchmarks, AMULET 1.5 [27], DYPOSUB [42] and REVSCA-2.0 [40] almost always solve around the same amount of benchmarks, where AMULET 1.5 is an order of magnitude faster than related work. Additionally AMULET 1.5 is the only tool which allows to generate proof certificates. DYPOSUB and REVSCA-2.0 outperform related work on optimized benchmarks, which are closer to industrial multipliers.

References

1. L. Amarú, P.-E. Gaillardon, and G. De Micheli. The EPFL combinational benchmark suite. In *International Workshop on Logic & Synthesis (IWLS)*, pages 57–61, 2015.
2. L. Amarú, P.-E. Gaillardon, and G. De Micheli. The EPFL combinational benchmark suite. https://www.epfl.ch/labs/lsi/page-102566-en-html/benchmarks/, 2020.
3. L. Amarú, P.-E. Gaillardon, and G. De Micheli. The EPFL combinational benchmark suite. https://github.com/lsils/benchmarks, 2020.
4. P. Beame, R. Impagliazzo, J. Krajícek, T. Pitassi, and P. Pudlák. Lower Bounds on Hilbert's Nullstellensatz and Propositional Proofs. In *Proc. London Math. Society*, volume s3-73, pages 1–26, 1996.
5. P. Beame and V. Liew. Towards Verifying Nonlinear Integer Arithmetic. In *International Conference on Computer Aided Verification, CAV 2017*, volume 10427 of *LNCS*, pages 238–258. Springer, 2017.
6. Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/ alanmi/abc/, 2019. Bitbucket Version 1.01.
7. A. Biere. Collection of Combinational Arithmetic Miters Submitted to the SAT Competition 2016. In *SAT Competition 2016*, volume B-2016-1 of *Dep. of Computer Science Report Series B*, pages 65–66. University of Helsinki, 2016.
8. A. Biere. Weaknesses of CDCL solvers, August 2016. In Fields Institute Workshop on Theoretical Foundations of SAT Solving, http://www.fields.utoronto.ca/talks/weaknesses-cdcl-solvers.
9. A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Dep. of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.
10. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.
11. R. E. Bryant and Y. Chen. Verification of arithmetic circuits using binary moment diagrams. *STTT*, 3(2):137–155, 2001.
12. B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, 1965.
13. Y. Chen and R. E. Bryant. Verification of Arithmetic Circuits with Binary Moment Diagrams. In *Design Automation Conference, DAC 1995*, pages 535–541. ACM, 1995.
14. Y. Chen, E. Clarke, P. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao. Verification of All Circuits in a Floating-Point Unit Using Word-Level Model Checking. In *FMCAD 1996*, volume 1166 of *LNCS*, pages 19–33. Springer, 1996.
15. M. J. Ciesielski, T. Su, A. Yasin, and C. Yu. Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model. *IEEE TCAD*, pages 1–1, 2019. Early acces.
16. M. J. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi. Verification of Gate-level Arithmetic Circuits by Function Extraction. In *Design Automation Conference, DAC 2015*, pages 52:1–52:6. ACM, 2015.
17. M. Clegg, J. Edmonds, and R. Impagliazzo. Using the Groebner Basis Algorithm to Find Proofs of Unsatisfiability. In *STOC 1996*, pages 174–183. ACM, 1996.
18. D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms*. Springer-Verlag New York, 1997.
19. M. J. H. Heule and A. Biere. Proofs for Satisfiability Problems. In *All about Proofs, Proofs for All Workshop, APPA 2014*, volume 55, pages 1–22. College Publications, 2015.
20. N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi. Formal Design of Arithmetic Circuits Based on Arithmetic Description Language. *IEICE Transactions*, 89-A(12):3500–3509, 2006.
21. Homma Laboratory, RIEC, Tohoku University. Arithmetic Module Generator. https://www.ecsis.riec.tohoku.ac.jp/topics/amg/.
22. W. A. Hunt, Jr., M. Kaufmann, J. Strother Moore, and A. Slobodova. Industrial hardware and software verification with ACL2. *Philos. Trans. Royal Soc. A*, 375(2104):20150399, 2017.

23. D. Kaufmann. AMulet 1.5. https://github.com/d-kfmnn/amulet, 2020.
24. D. Kaufmann. *Formal Verification of Multiplier Circuits using Computer Algebra*. PhD thesis, Informatik, Johannes Kepler University Linz, 2020.
25. D. Kaufmann and A. Biere. Nullstellensatz-proofs for multiplier verification. In *CASC*, Lecture Notes in Computer Science. Springer, 2020. To appear.
26. D. Kaufmann, A. Biere, and M. Kauers. Incremental Column-wise verification of arithmetic circuits using computer algebra. *Formal Methods in System Design*, 2019. Online first.
27. D. Kaufmann, A. Biere, and M. Kauers. Verifying Large Multipliers by Combining SAT and Computer Algebra. In *FMCAD 2019*, pages 28–36. IEEE, 2019.
28. D. Kaufmann, M. Fleury, and A. Biere. Pacheck and Pastèque, Checking Practical Algebraic Calculus Proofs. In *FMCAD 2020*, volume 1 of *FMCAD*, pages 264–269. TU Vienna Academic Press, 2020.
29. D. Kaufmann, M. Kauers, A. Biere, and D. Cok. Arithmetic Verification Problems Submitted to the SAT Race 2019. In *SAT Race 2019*, volume B-2019-1 of *Dep. of Computer Science Report Series B*, page 49. University of Helsinki, 2019.
30. M. Kaufmann and J. S. Moore. ACL2 Version 8.2. http://www.cs.utexas.edu/users/moore/acl2/, 2019.
31. A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE TCAD*, 21(12):1377–1394, 2002.
32. V. Liew, P. Beame, J. Devriendt, J. Elffers, and J. Nordström. Verifying Properties of Bit-vector Multiplication Using Cutting Planes Reasoning. In *FMCAD 2020*, volume 1 of *FMCAD*, pages 194–204. TU Vienna Academic Press, 2020.
33. J. Lv and P. Kalla. Formal Verification of Galois Field Multipliers Using Computer Algebra Techniques. In *International Conference on VLSI Design, VLSID 2012*, pages 388–393. IEEE Computer Society, 2012.
34. J. Lv, P. Kalla, and F. Enescu. Efficient Gröbner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits. *IEEE TCAD*, 32(9):1409–1420, 2013.
35. A. Mahzoon, D. Große, and R. Drechsler. GenMul. http://sca-verification.org/genmul, 2018.
36. A. Mahzoon, D. Große, and R. Drechsler. GenMul. https://github.com/amahzoon/genmul, 2018.
37. A. Mahzoon, D. Große, and R. Drechsler. PolyCleaner. http://sca-verification.org/polycleaner, 2018.
38. A. Mahzoon, D. Große, and R. Drechsler. PolyCleaner: Clean your Polynomials before Backward Rewriting to verify Million-gate Multipliers. In *ICCAD 2018*, pages 129:1 – 129:8. ACM, 2018.
39. A. Mahzoon, D. Große, and R. Drechsler. RevSCA and RevSCA-2.0. http://sca-verification.org/revsca, 2019.
40. 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*, pages 185:1–185:6. ACM, 2019.
41. A. Mahzoon, D. Große, C. Scholl, and R. Drechsler. DyPoSub. http://sca-verification.org/dyposub, 2020.
42. A. Mahzoon, D. Große, C. Scholl, and R. Drechsler. Towards formal verification of optimized and industrial multipliers. In *DATE*, pages 544–549. IEEE, 2020.
43. A. Niemetz, M. Preiner, C. Wolf, and A. Biere. Btor2 , BtorMC and Boolector 3.0. In *CAV 2018*, volume 10981 of *LNCS*, pages 587–595. Springer, 2018.
44. B. Parhami. *Computer Arithmetic - Algorithms and Hardware designs*. Oxford University Press, 2000.
45. E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, O. Wienand, and E. Karibaev. Modeling of Custom-Designed Arithmetic Components for ABL Normalization. In *Forum on Specification and Design Languages, FDL 2008*, pages 124–129. IEEE, 2008.
46. D. Ritirc, A. Biere, and M. Kauers. A Practical Polynomial Calculus for Arithmetic Circuit Verification. In *SC2 2018*, pages 61–76. CEUR-WS, 2018.

47. E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS. https://ptolemy.berkeley.edu/projects/embedded/pubs/downloads/sis/index.htm.
48. E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, May 1992.
49. H. Sharangpani and M. L. Barton. Statistical analysis of floating point flaw in the pentium processor. 1994.
50. D. Stoffel and W. Kunz. Equivalence checking of arithmetic circuits on the arithmetic bit level. *IEEE TCAD*, 23(5):586–597, 2004.
51. M. Temel. MultGen. https://github.com/temelmertcan/multgen, 2020.
52. M. Temel, A. Slobodová, and W. A. Hunt. Automated and scalable verification of integer multipliers. In *CAV (1)*, volume 12224 of *Lecture Notes in Computer Science*, pages 485–507. Springer, 2020.
53. 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. *IEEE Trans. Comput.*, 56(10):1401–1414, 2007.
54. O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G. Greuel. An Algebraic Approach for Proving Data Correctness in Arithmetic Data Paths. In *International Conference on Computer Aided Verification, CAV 2008*, volume 5123 of *LNCS*, pages 473–486. Springer, 2008.
55. C. Wolf. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.
56. C. Yu. Algebraic RewriTing in ABC. https://github.com/ycunxi/abc, 2018.
57. C. Yu, W. Brown, D. Liu, A. Rossi, and M. J. Ciesielski. Formal Verification of Arithmetic Circuits by Function Extraction. *IEEE TCAD*, 35(12):2131–2142, 2016.
58. C. Yu and M. J. Ciesielski. Efficient Parallel Verification of Galois Field Multipliers. In *Asia and South Pacific Design Automation Conference, ASP-DAC 2017*, pages 238–243. IEEE, 2017.
59. C. Yu and M. J. Ciesielski. Formal Analysis of Galois Field Arithmetic Circuits-Parallel Verification and Reverse Engineering. *IEEE TCAD*, 38(2):354–365, 2019.
60. C. Yu, M. J. Ciesielski, and A. Mishchenko. Fast Algebraic Rewriting Based on And-Inverter Graphs. *IEEE TCAD*, 37(9):1907–1911, 2018.