International Journal on Software Tools for Technology Transfer (2023) 25:133–144
https://doi.org/10.1007/s10009-022-00688-6

GENERAL

Special Issue: TACAS 2021

Improving AMULET2 for verifying multiplier circuits using SAT solving and computer algebra

Daniela Kaufmann[1] · Armin Biere[2]

Accepted: 4 November 2022 / Published online: 11 January 2023

Abstract
Verifying arithmetic circuits and most prominently multiplier circuits is an important problem which in practice is still considered to be challenging. One of the currently most successful verification techniques relies on algebraic reasoning. In this article, we present AMULET2, a fully automatic tool for verification of integer multipliers combining SAT solving and computer algebra. Our tool models multipliers given as and-inverter graphs as a set of polynomials and applies preprocessing techniques based on elimination theory of Gröbner bases. Finally, it uses a polynomial reduction algorithm to verify the correctness of the given circuit. AMULET2 is a re-factorization and improved re-implementation of our previous verification tool AMULET1 and cannot only be used as a stand-alone tool but also serves as a polynomial reasoning framework. We present a novel XOR-based slicing approach and discuss improvements on the data structures including monomial sharing.

Keywords Circuit verification · Multipliers · Computer algebra · SAT solving · Gröbner basis · Proof certificates

1 Introduction

Formal verification of arithmetic circuits is important to prevent issues like the infamous Pentium FDIV bug [37]. Up to now, there have been many attempts to verify these circuits, but even today, the problem of fully automatic verification of arithmetic circuits, and especially multipliers, is still considered to be hard.

Methods based on decision diagrams [6] rely on manual structural decomposition of the multiplier. Approaches based on satisfiability checking (SAT) do not scale [3]. Recently, progress has been made using theorem provers [39]. However, the multipliers have to be given as hierarchical SVL netlists, which rely on preservation of information of the circuits. For flattened gate-level multipliers, the currently most successful technique is based on algebraic reasoning [8,17,23,34,35]. In this line of work, the circuit is modeled as a set of polynomials that generate a Gröbner basis and the specification is then checked to be implied by the circuit polynomials using a polynomial reduction algorithm.

In our approach [23], we apply a combination of SAT solving and computer algebra. Certain parts of the multiplier, i.e., complex final stage adders that are generate-and-propagate adders [36] are hard to verify using computer algebra, but are easy to verify using SAT solvers [29]. Therefore, we apply adder substitution [23] and replace complex final stage adders by simple ripple-carry adders that can be verified using computer algebra. The equivalence of the adders is verified using SAT solvers. The correctness of the simplified multiplier is shown using computer algebra [23].

This article presents our tool AMULET2, a successor of AMULET1 [23,26]. The version history of AMULET is depicted in Table 1. AMULET2 reads multipliers given as and-inverter graphs [30] and fully automatically applies adder substitution and verifies the (simplified) circuit. However, the verification process might not be error-free. In order to validate the verification results algebraic proof certificates that monitor the verification process can be generated in AMULET2, which can be checked using, e.g., PACHECK [19].

This work is supported by the LIT AI Lab funded by the State of Upper Austria.

✉ Daniela Kaufmann
daniela.kaufmann@tuwien.ac.at

Armin Biere
biere@informatik.uni-freiburg.de

1 TU Wien, Vienna, Austria

2 Albert-Ludwigs-University, Freiburg, Germany

Table 1 Version history of AMULET

Version	Reference	Features
AMULET1.0	[23,26]	
AMULET1.5	[15]	improved adder substitution
AMULET2.0	[22]	modular re-implementation
AMULET2.1	This article	reduced memory usage

AMULET2 is a modular C++ re-implementation of AMULET1 (while AMULET1 consists of a single C file). AMULET2 is not only a stand-alone tool but also serves as a reasoning framework, i.e., parts can easily be integrated into different workflows. AMULET2 still provides the same functionality as AMULET1, but with improved algorithms, based on the same theory [17,23]. In this article, we focus on the novelties of AMULET2.

This article extends and revises work presented in a tool demonstration paper at TACAS'21 [22]. We extend the preliminaries in Sect. 2 and give a self-contained introduction to bit-level verification using computer algebra following [23]. Additionally, we expand our discussion on design decisions in AMULET2 and present our novel XOR-based slicing approach in Sect. 3 in more detail. Section 4 discusses AMULET2.1, which improves the memory usage of AMULET2.0 [22]. We further extend the experimental evaluation in Sect. 6.

2 Circuit verification using computer algebra

In this section, we introduce multiplier circuits and their architectural details. We present the algebraic concepts that are needed in the technique of automated circuit verification using computer algebra and discuss adder substitution and proof certificates.

2.1 Multiplier circuits

A digital circuit implements a logical function and computes binary digital values, given binary values at the inputs. Multipliers are circuits that compute the product of two-input bit vectors. The computation is typically 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.

In this article, we consider gate-level integer multipliers without latches with $2n$ 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 inputs the specification $\mathcal{U}_n = 0$ holds, where

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

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. A signed multiplier is correct if and only if for all inputs the specification $\mathcal{S}_n = 0$ holds, where

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

A common representation of digital circuits is the encoding as an and-inverter graph (AIG) [30]. 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 usually contains more nodes, than the gate-level representation, but has an unequivocal syntax and semantics, and is very efficient to manipulate. Since an AIG is a directed graph, we are able to determine for each gate g in the circuit its *input cone*, i.e., the set of gates g_i for which there exists a path from g to g_i.

The space and time complexity of a multiplier depends on its architecture. In general, a multiplier circuit can be divided into three parts [36]. 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 [36].

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. The well-known accumulation structures are for example array or diagonal accumulation, Wallace trees, or compressor trees [36].

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 compute the carry bits alongside the sum bits using sequences of OR gates. They are called *generate-and-propagate* (GP) adders. Examples are carry-lookahead adders, Ladner-Fischer adders, Han-Carlson adders, and Kogge–Stone adders [36].

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

2.2 Algebra

Let us now briefly summarize algebraic concepts. Our algebraic setting follows [9] and we assume $0 \in \mathbb{N}$.

- Let X denote the set of variables $\{x_1, \ldots, x_l\}$. By $\mathbb{Z}[X]$, we denote the ring of polynomials in variables X with coefficients in $\mathbb{Z}$.
- A *term* $\tau = x_1^{d_1} \cdots x_l^{d_l}$ is a product of powers of variables for $d_i \in \mathbb{N}$. A *monomial* is a multiple of a term $c\tau$ with $c \in \mathbb{Z} \setminus \{0\}$ and a *polynomial* is a finite sum of monomials with pairwise distinct terms.
- On the set of terms, an order $\leq$ is fixed such that for all terms τ, σ_1, σ_2, it holds that $1 \leq \tau$ and further $\sigma_1 \leq \sigma_2 \Rightarrow \tau\sigma_1 \leq \tau\sigma_2$. One such order is the so-called *lexicographic term order*, defined as follows. If the variables of a polynomial are ordered $x_1 > x_2 > \ldots x_l$, then given two distinct terms $\sigma_1 = x_1^{d_1} \cdots x_l^{d_l}$, $\sigma_2 = x_1^{e_1} \cdots x_l^{e_l}$ it holds $\sigma_1 < \sigma_2$ if and only if there exists an index i with $d_j = e_j$ for all $j < i$, and $d_i < e_i$.
- For a polynomial $p = c\tau + \cdots$, the largest term τ (w.r.t. $\leq$) is called the *leading term* $\text{lt}(p) = \tau$. The *leading coefficient* $\text{lc}(p) = c$ and *leading monomial* $\text{lm}(p) = c\tau$ are defined accordingly.

Definition 1 A set $I \neq \{\} \subseteq \mathbb{Z}[X]$ is called an *ideal* if $\forall\, p, q \in I : p + q \in I$ and $\forall\, p \in \mathbb{Z}[X]\, \forall\, q \in I : pq \in I$. If $I \subseteq \mathbb{Z}[X]$ is an ideal, then a set $P = \{p_1, \ldots, p_m\} \subseteq \mathbb{Z}[X]$ is called a *basis* of I if $I = \{q_1 p_1 + \cdots + q_m p_m \mid q_1, \ldots, q_m \in \mathbb{Z}[X]\}$. We say I *is generated by* P and write $I = \langle P \rangle$.

The theory of D-Gröbner bases [2] offers a unique decision procedure for the so-called ideal membership problem over $\mathbb{Z}[X]$, i.e., given $q \in \mathbb{Z}[X]$ and a basis $P = \{p_1, \ldots, p_m\} \subseteq \mathbb{Z}[X]$, decide whether q belongs to the ideal generated by P. We discuss in Sect. 2.3 why we choose the ring $\mathbb{Z}[X]$, instead of a polynomial ring where the coefficient domain is a field, which would involve the standard theory of Gröbner bases [7].

Let $p, q, r \in \mathbb{Z}[X]$ and $P \subseteq \mathbb{Z}[X]$. Some facts about the theory of D-Gröbner bases over Euclidean domains, such as $\mathbb{Z}$, are

- A basis P of an ideal $I \subseteq \mathbb{Z}[X]$ is called a *D-Gröbner basis* of I if and only if $\forall q \in I\, \exists p \in P : \text{lm}(p) \mid \text{lm}(q)$.
- Every ideal of $\mathbb{Z}[X]$ has a D-Gröbner basis, and there is an algorithm [2, Thm 10.14] that, given an arbitrary basis of an ideal, computes a D-Gröbner basis of it in finitely many steps. It is based on repeated computation of so-called *S-polynomials* spol and *G-polynomials* gpol. The precise definition of spol and gpol is not important for our application and thus is not included in this article.
- We say q *D-reduces* to r w.r.t. p if there exists a monomial m' in q with $m' = m\,\text{lm}(p)$ and $r = q - mp$. If $m' = \text{lm}(q)$, we say *top-D-reduction*.
- The remainder r of the D-reduction of q by P is such that $q - r \in \langle P \rangle$ and r is *D-reduced* w.r.t. P.
  If r is calculated using only top-D-reductions, then we say r is *top-D-reduced* w.r.t. P.
- (Cor. 10.12 in [2]) A set $P \subseteq D[X]$ is a D-Gröbner basis of $\langle P \rangle$ if and only if for all pairs $(p_1, p_2) \in P \times P$, the remainder of D-reducing $\text{spol}(p_1, p_2)$ w.r.t. P is zero and $\text{gpol}(p_1, p_2)$ top-D-reduces to zero w.r.t. P.
- (Thm. 10.23 in [2]) Let P be a D-Gröbner basis. Then, $q \in \langle P \rangle \Leftrightarrow q$ D-reduces to 0 w.r.t. P.

In general, the construction of D-Gröbner basis is very expensive. However, there exist some results which state when the computation of spol and gpol is superfluous when computing a D-Gröbner basis. We will heavily use the following lemma in our application.

Lemma 1 *(Thm. 11 in [31]) Let* $p_1, p_2 \in \mathbb{Z}[X]$ *be such that* $\text{lcm}(\text{lt}(p_1), \text{lt}(p_2)) = \text{lt}(p_1)\,\text{lt}(p_2)$. *If* $\text{lc}(p_1) \mid \text{lc}(p_2)$ *then* $\text{spol}(p_1, p_2)$ *and* $\text{gpol}(p_1, p_2)$ *top-D-reduce to zero.*

2.3 Circuit verification using computer algebra

As discussed previously let $a_0, \ldots, a_{n-1}$, $b_0, \ldots, b_{n-1}$, $s_0, \ldots, s_{2n-1} \in \{0, 1\}$ denote the input and output bits of the given multiplier and let $l_1, \ldots, l_k \in \{0, 1\}$ denote the internal AIG nodes. Hence, $X = \{a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, l_1, \ldots, l_k, s_0, \ldots, s_{2n-1}\}$.

The semantics of each AIG node, cf. Fig. 1, implies a polynomial relation among the input and output variables, such as the following ones:

$$\begin{array}{lll} u = v \wedge w & \text{implies} & -u + vw = 0 \\ u = v \wedge \neg w & \text{implies} & -u - vw + v = 0 \\ u = \neg v \wedge \neg w & \text{implies} & -u + vw - v - w + 1 = 0 \end{array}$$

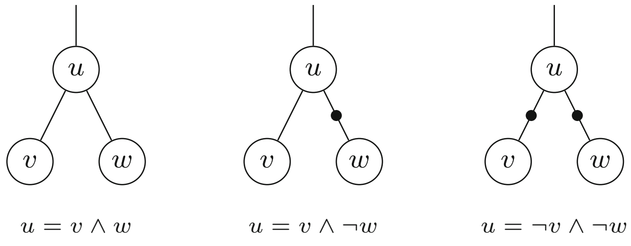

Fig. 1 Polynomial encodings implied by AIG nodes

The polynomials are chosen such that the Boolean roots of the polynomials are the solutions of the node constraints and vice versa. Let $G(C) \subseteq \mathbb{Z}[X]$ be the set of polynomials that contains for each AIG node its corresponding polynomial relation. All variables $x \in X$ are Boolean and we enforce this property by the set of *Boolean value constraints* $B(X) = \{x(1-x) \mid x \in X\} \subseteq \mathbb{Z}[X]$. As the right side of all polynomial equations is always zero, we write f instead of $f = 0$.

The polynomials in $G(C)$ are ordered according to a lexicographic order, based on a variable ordering such that the output variable of a node is always greater than the inputs of the gate [32]. This ordering is also called *reverse topological term ordering*.

Definition 2 Let $P \subseteq \mathbb{Z}[X]$. If for a certain term order, all leading terms of P only consist of at most a single variable with exponent 1 and are unique and further $\operatorname{lc}(p) \in \{-1, 1\}$ for all $p \in P$, then we say P has *unique monic leading terms* (UMLT). Let $X_0(P) \subseteq X$ be the set of all variables that do not occur as leading terms in P. We further define $B_0(P) = B(X_0(P))$.

Example 1 The set $P = \{-x - y + 1, -y + z\} \subseteq \mathbb{Z}[x, y, z]$ has UMLT for the lexicographic term order $x > y > z$. Correspondingly, $X_0(P) = \{z\}$ and $B_0(P) = \{-z^2 + z\}$.

In the following, these $X_0(P)$ will represent inputs of a circuit, and accordingly, $B_0(P)$ are the Boolean value constraints only on its inputs. Let $J(C) = \langle G(C) \cup B_0(X)\rangle \subseteq \mathbb{Z}[X]$ be the ideal generated by $G(C) \cup B_0(X)$.

Additionally, we add the constant 2^n to the ideal generators of $J(C)$, because this admits modular reasoning and allows us to eliminate monomials with too large coefficients from polynomials [23]. Adding constants is only possible in $\mathbb{Z}[X]$. If we would choose the ring $\mathbb{Q}[X]$ as our polynomial domain, we are able to deduce from $2^n \in J(C)$, that $\frac{1}{2^n}2^n = 1 \in J(C)$, thus making any verification attempt obsolete. In the ring $\mathbb{Z}[X]$, this deduction is not possible, because $\frac{1}{2^n} \notin \mathbb{Z}[X]$.

The circuit fulfills its specification if and only if we can derive that $\mathcal{L} \in J(C)$ [23]. The following theorem shows that we do not have to compute a D-Gröbner basis for $J(C)$.

Theorem 1 *(Thm. 5 of* [23]*) Let* $J(C) = \langle G(C) \cup B_0(C) \cup \{2^n\}\rangle \subseteq \mathbb{Z}[X]$. *Then,* $G(C) \cup B_0(C) \cup \{2^n\}$ *is a D-Gröbner basis of* $J(C)$ *w.r.t. to a fixed reverse topological term ordering.*

Proof Since $G(C)$ has UMLT, $G(C) \cup B_0(C) \cup \{2^n\}$ has UMLT and thus it holds by Lemma 1, top-D-reduction of $\operatorname{spol}(p, q)$ and $\operatorname{gpol}(p, q)$ by $\{p, q\}$ gives the remainder zero for any choice $p, q \in G(C) \cup B_0(C) \cup \{2^n\}$.

Hence, the correctness of the circuit can be established by D-reducing $\mathcal{L}$ by the polynomials $G(C) \cup B_0(X) \cup \{2^n\}$ and checking whether the result is zero. Because of the UMLT property, D-reduction actually boils down to simple substitutions of the leading terms.

The considered logical gates are functional, given values of the inputs $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1} \in \{0, 1\}$, all values of $l_1, \ldots, l_k, s_0, \ldots, s_{2n-1}$ are either 0 or 1. Thus, we derive that $B(X) \subseteq J(C)$.

Hence, after each D-reduction step of $\mathcal{L}$ by a polynomial $g \in G(C)$, we reduce the result by $B(X) \cup \{2^n\}$, i.e., reduce all exponents greater than one to one and reduce all monomials with coefficients that are greater than $2^n - 1$. This step helps to reduce the size of the intermediate reduction results.

If the final remainder r of D-reducing $\mathcal{L}$ by $G(C) \cup B(X) \cup \{2^n\}$ is not zero, we know that the circuit contains an error and we are able to derive at least one concrete counter example for which the circuit computes wrong results. By the choice of the term order, r only contains input variables $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}$, and none of them appears with degree greater than one. Let m be a monomial of r with a minimal number of variables, which includes the case where m is constant. Since exponents are at most one, the set of variables of monomials in r differ by at least one variable.

Now choose a_i (b_j) to evaluate to 1 if and only if $a_i \in m$ $(b_j \in m)$. By this choice, all monomials of r except m vanish (evaluate to zero). Thus, r evaluates to the (nonzero) coefficient of m, contradicting the specification.

Variable Elimination Several experiments have shown that simply reducing the specification by $G(C) \cup B(X) \cup \{2^n\}$ leads to large intermediate results [17,33]. Hence, we eliminate variables in $G(C)$ prior to reduction to yield a more compact D-Gröbner basis [23]. In the preprocessing step, we repeatedly select and remove variables $v \in X \setminus X_0$ from $G(C)$. Let $p_v \in G(C)$ such that $\operatorname{lt}(p_v) = v$. Since $G(C)$ has UMLT and $v \notin X_0$, such a p_v exists. All polynomials $g \neq p_v \in G(C)$ that contain v are being D-reduced by p_v, thus only p_v will contain v. Since $v = \operatorname{lt}(p_v)$, we can safely remove p_v from $G(C)$ without violating the D-Gröbner basis property [23].

Example 2 Let $G(C) = \{-x - y + 1, -y + z\} \subseteq \mathbb{Z}[x, y, z]$ and we remove v, i.e., $p_v := -y + z$. D-reducing $-x - y + 1$ by p_v gives $-x + z - 1$. We remove p_v from $G(C)$ and obtain $G(C) = \{-x - z + 1\} \subseteq \mathbb{Z}[x, z]$.

2.4 Adder Substitution

Certain parts of the multiplier, i.e., when the FSA is a GP adder, are hard to verify using computer algebra, even with prior variable elimination. In a GP adder with inputs $x_0, \ldots, x_m, y_0, \ldots, y_m, c_{in}$ and outputs $s'_0, \ldots, s'_m, c_{out}$ the output bits s'_i are calculated as $s'_i = p_i \oplus c_i$, with $p_i = x_i \oplus y_i$. The carries c_i are recursively generated using the equation $c_i = (x_{i-1} \wedge y_{i-1}) \vee (c_{i-1} \wedge p_{i-1})$ with $c_{m+1} = c_{out}$ and

$c_0 = c_{in}$. The precise derivation of the carries c_i (recursively, unrolled or mixed) depends on the architecture of the adders, but is generally computed using sequences of OR gates. These sequences of OR gates make the GP adders hard to verify using the algebraic approach as the following example shows.

Example 3 Let $o = x_1 \vee x_2 \vee \ldots \vee x_n$ represent a sequence of n OR gates. Since $o = x_1 \vee x_2 \vee \ldots \vee x_n \Leftrightarrow \neg o = \neg x_1 \wedge \neg x_2 \wedge \ldots \wedge \neg x_n$, we derive the corresponding algebraic representation $o = 1-(1-x_1)(1-x_2)\ldots(1-x_n)$ consisting of $2^n - 1$ monomials.

In AMULET, we identify whether the FSA is a GP adder, using the equations $s'_i = p_i \oplus c_i$ and $p_i = x_i \oplus y_i$. If we detect that the FSA is a GP adder, we substitute the FSA by a simple RC adder, which has the same inputs as the original FSA. The first two stages, PPG and PPA, are not changed. We generate a bit-level miter in conjunctive normal form (CNF) to prove that the ripple-carry adder is equivalent to the GP adder. The miter is verified by a SAT solver. However, if the FSA is not a GP adder, we do not apply adder substitution. After substitution, we verify the rewritten AIG in AMULET using computer algebra.

2.5 Proof certificates

The verification process itself might not be error-free. A common technique to guarantee the correctness of the result is to generate simple proof certificates, which monitor the steps of the verification process and enable reproducing the proof. These certificates can be checked by stand-alone proof checkers.

Algebraic proof systems reason about polynomial equations. Given a set of polynomials G, the aim is to show that an equation $f = 0$ is implied by G, i.e., $f \in \langle G \rangle$, using a sequence of proof steps. In our application, f is the specification $\mathcal{L}$ of a multiplier, and the polynomials g_i are the gate constraints $G(C)$ and Boolean value constraints $B(X)$. AMULET2 supports the practical algebraic calculus (PAC) [27], like AMULET1, but now also Nullstellensatz proofs [1,21]. Examples for both formats are depicted in Example 4.

PAC proofs consist of a sequence of steps that either encode polynomial addition or multiplication. During proof checking in, e.g., PACHECK [27], each proof step is checked for correctness. Thus, PAC proofs allow the user to localize possible errors. The Boolean value constraints are handled implicitly, i.e., the proof checker internally calculates $x \cdot x = x$.

In the Nullstellensatz proof format, co-factors $\alpha_i \in \mathbb{Z}[X]$ are provided such that there exist $\beta_j \in \mathbb{Z}[X]$ with $\mathcal{L} = \sum_{g_i \in G(C)} \alpha_i g_i + \sum_{h_j \in B(X)} \beta_j h_j + \gamma 2^n$. This proof format is very concise, as only the ordered list of co-factors α_i, γ is printed as a proof certificate. The Boolean value constraints are treated implicitly during proof checking, thus it is not necessary to provide co-factors β_j. However, proof checking consists of calculating one huge linear combination, making it nearly impossible to locate a possible error in the certificate.

Example 4 Let $z = x \wedge y$, $x = a \wedge b$, and $y = a \oplus b$, with the specification $z = 0$. The algebraic encoding of the gates is

```
1 -z+xy;
2 -x+ab;
3 -y+a+b-2ab;
```

A proof in PAC is as follows, using labels:

```
4 * 3, -ab, yab;     //(-y+a+b-2ab)*(-ab)= yab
5 * 2, -y, xy-yab;   //(-x+ab)*(-y)= xy-yab
6 + 4, 5, xy;        //(yab)+(xy-yab)= xy
7 * 1, -1, z-xy;     //(-z+xy)*(-1)= z-xy
8 + 6, 7, z;         //(xy)+(z-xy)= z
```

A proof in Nullstellensatz is `-1, -y, -ab`. Because $(-1)(-z+xy) - y(-x+ab) - ab(-y+a+b-2ab) = z$, after reducing all exponents.

3 AMULET2

We present the architecture of AMULET2 and discuss novel optimizations. The published version of our tool AMULET2 including its artifact is available at [16]. The maintained version is available at [18] and is published as open source under the MIT license.

The architecture of AMULET2 is shown in Fig. 2. The blue arrows outside the box reflect inputs and outputs of AMULET2. The gray arrows inside the box depict the dependencies of the individual modules within AMULET2.

In contrast to AMULET1, which consists of a single C file, AMULET2 is split into components, which allows integrating only parts in different workflows. For example, the artifact [16] contains two demos, where (i) the *Polynomial Library* is used for simple polynomial calculations and (ii) the *Polynomial Solver* is used for verification of multipliers that are not given as AIGs.

AMULET2 is implemented in C++11 and consists of around 3 800 lines of code. It relies on the AIGER library [5] to process the given AIG and the GMP library [11] to represent large integers.

AMULET2 supports three modes, *substitute*, *verify*, and *certify*. The mode of AMULET2 is triggered by the command line input, see also the usage demo in Sect. 5. In substitution mode, AMULET2 parses the AIG, allocates the internal gate structure, and invokes the substitution engine for adder substitution. In verification mode, AMULET2 reads the AIG and initializes the gate structure. Afterward, the circuit is verified

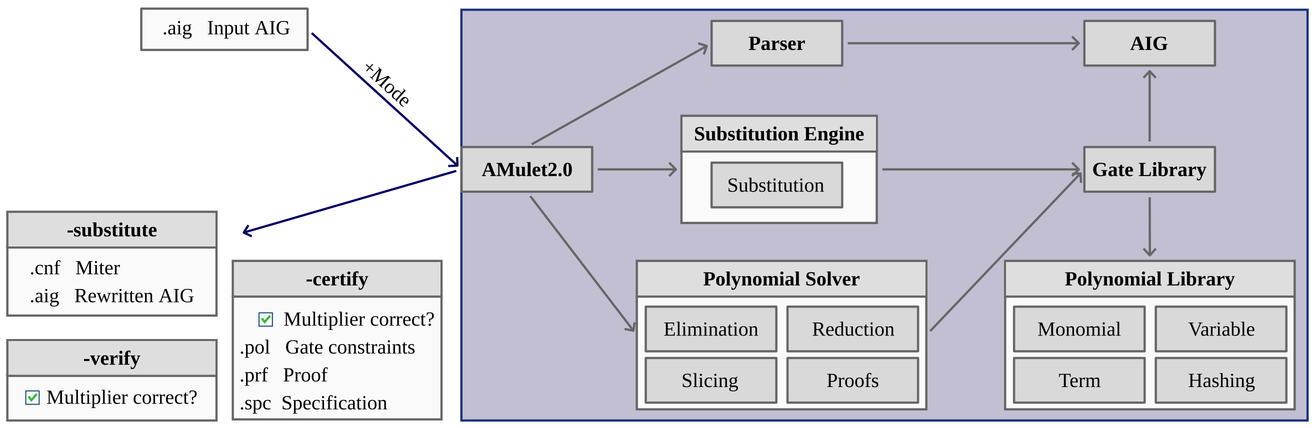

Fig. 2 Tool architecture of AMULET2

in the polynomial solver using polynomial operations of the polynomial library. In certification mode, proofs are generated in addition. All heuristics and optimizations presented in Sects. 2.3 and 2.4 can only lead to an increase in the runtime of our tool, but do not affect its soundness and completeness.

In the following, we present the individual components of AMULET2 and summarize the improvements over AMULET1.

3.1 Parser module

AMULET2 checks whether the given AIG circuit fulfills the requirements described in Sect. 2, i.e., the AIG circuit has an even number of inputs and an equal number of outputs and does not contain any latches. The AIG module wraps functions of the external AIGER library that are needed to process the input file.

3.2 Gate library

After parsing, AMULET2 allocates a gate for each AIG node, which includes structural information of the node, such as dependencies, or whether the gate represents an input/output or an XOR gate. Each gate is linked to a unique variable. If the given AIG is verified or certified, AMULET2 also initializes the gate constraints and creates the specification polynomial $\mathcal{L} \in \mathbb{Z}[X]$.

3.3 Substitution engine

In substitution mode, AMULET2 applies heuristic pattern matching to identify GP adders [23]. The algorithms in the substitution engine are almost the same as in AMULET1 [26] and highly relate on the structure of GP adders, cf. Sect. 2.4. In particular, we use the fact that the outputs of GP adders are always outputs of XOR gates, but the carries are never outputs of XOR gates. This allows us within one iteration over the output bits to mark the carries c_i (including the carry output c_{out}) and propagate bits p_i of the GP adder and we are further able to identify and mark all inputs x_i, y_i of the GP adder using the relation $p_i = x_i \oplus y_i$.

In a second step, we mark all gates that belong to the FSA. We start at the carry output c_{out} resp. sum outputs and follow all paths in the input cones until we either reach a marked input x_i, y_i, or c_{in}. We mark the visited variables. If at some point we reach one of the input variables a_i, b_j of the multiplier, the FSA is not a GP adder, i.e., we were not able to clearly identify the boundaries of the FSA. Consequently, adder substitution was not successful and the initially given AIG is returned without generating a bit-level miter. If on the other hand all paths stop at the marked inputs or at c_{in}, we have successfully identified and marked all gates belonging to a GP adder and apply adder substitution.

In AMULET2, we enhanced the identification heuristics and cover special cases, e.g., in some architectures, we have $x_i = x_{i+1}$, that are not considered in AMULET1. Thus, AMULET2 is able to detect more GP adders. After a positive GP pattern match, AMULET2 replaces the GP adder by an equivalent RC adder. A bit-level miter is generated in CNF to verify the equivalence of the adders. The rewritten multiplier and the CNF miter are printed to the provided output files.

3.4 Polynomial solver

The polynomial solver is a modernized version of the solving engine of AMULET1 [26] and is used to verify or certify the given multiplier. In a nutshell, the polynomial solver first applies preprocessing by eliminating selected variables. Afterward, the remaining variables are ordered into column-wise slices, such that we can apply our incremental verification algorithm [25], where we split the specification $\mathcal{L}$ into multiple polynomials and verify the multiplier by deriving the correctness of each slice using polynomial reduction.

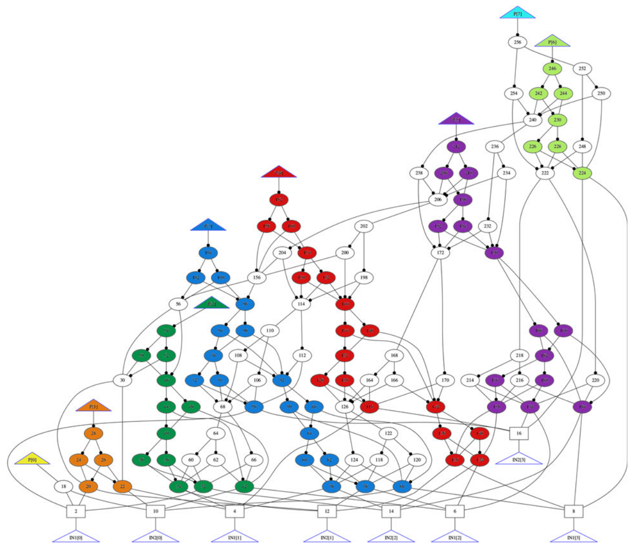

Fig. 3 XOR-skeletons in a simple 4-bit multiplier

The necessary polynomial operations are implemented in our **Polynomial Library** described below.

In AMULET2, we eliminate variables before ordering them, whereas in AMULET1, it is the other way around. In a first step, we eliminate all internal gates of the XOR structures and all single-parent nodes in the AIG. Thus, fewer variables are considered for ordering, which improves computation time of AMULET2.

Furthermore, we include a novel XOR-based slicing approach in AMULET2, which relies on the fact that many multiplier architectures use XOR-skeletons to compute the output bits. We identify these skeletons as follows. We start from the output bits of the given multiplier and follow all paths as long as the gates are identified as (internal) XOR gates or encode partial products, i.e., both children are inputs of the AIG. All nodes of a skeleton are assigned to the same slice. Figure 3 shows the AIG of a simple 4-bit multiplier. We uniquely colored for each output bit the corresponding XOR-skeleton. Gates occurring between XOR-skeletons are assigned to the smaller (less significant) slice. Hence, after two iterations, all slices are fixed, which improves slicing compared to AMULET1. All variables that are not assigned to slices, e.g., gates used to compute the partial products in Booth encoding [36], are eliminated from the gate structure.

In few cases, we cannot identify XOR-skeletons, e.g., in multipliers containing a carry-select adder, and we fall back on the slicing approach of AMULET1 [26]: We slice based on input cones and eagerly move gates between slices to reduce the number of carries, by iterating multiple times over the variables.

After assigning gates to slices, AMULET2 reduces the slice-wise specifications incrementally by the sliced gate constraints and checks whether the final result is zero, similar to the implementation of AMULET1. If the final remainder is not zero, AMULET2 detects counter examples, i.e., input assignments for which the multiplier circuit computes an incorrect result.

In certification mode, AMULET2 tracks polynomial operations in the selected proof format and prints gate constraints, the generated proof, and the specification $\mathcal{L}$ to the provided files.

3.5 Polynomial library

The polynomial library implements the arithmetic operations for addition and multiplication of polynomials (by constants), and division by terms. Since all variables represent Boolean values, we always reduce exponents greater than one automatically to one, i.e., we assume $x \cdot x = x$. All algorithms for polynomial operations are optimized accordingly.

Polynomials are represented as sorted linked lists of monomials. Each monomial consists of a coefficient, represented using the GMP library, and a term. Terms are linked lists of variables, which are internally shared using a hash table.

In AMULET1, we do not share monomials and make hard copies in the few occasions when a monomial needs to be copied. This has the benefit that we can simply modify coefficients of the monomials, e.g., during addition. In our experiments, we observed that allocating new GMP objects is actually quite time consuming, and therefore, we now share monomials in AMULET2, using reference counting, which decreases verification time by a factor of two.

3.6 Improvements over AMULET1

AMULET2 contains some major improvements over AMULET1, which can be clearly seen in the experimental evaluation. At this point, we summarize the major improvements of AMULET2 over AMULET1 that are scattered in this section in one place:

- Modular re-implementation in C++, which allows reusing single components.
- Consideration of corner cases during adder substitution has the effect that more GP adders are detected.
- Restructuring the order of variable elimination and slicing to reduce the amount of gates that have to be ordered.
- Novel XOR-based slicing approach, which leads to faster verification time.
- Sharing of monomials in the data structure, which speeds up verification time.
- Invoking multiple proof formats.

4 AMULET2.1

Our experimental evaluation, cf. Sect. 6 shows that AMULET2.0 needs more memory than AMULET1 for 64-bit multipliers. We have addressed this issue and enhanced the algorithms and used data structures to reduce the memory usage. The improvements are published in AMULET2.1 [18], which further contains some bug-fixes on the slicing approach.

First of all, AMULET2.0 generates all gate constraints during the initialization phase. These polynomials are kept in memory through the whole verification process, although they might only be used in the final reduction steps. In AMULET2.1, we now generate the gate constraints on the fly. Whenever an original gate constraint is required during variable elimination or during reduction, we generate the corresponding polynomial encoding that is implied by the AIG. These polynomials will be deleted again after the elimination or the reduction step and we only keep the rewritten polynomials that are generated during variable elimination in memory, which reduces the size of the hash table used for sharing the terms/monomials significantly.

Although the deletion of gate constraints means that we sometimes have to generate the same polynomial equation multiple times, our experimental evaluation shows that this is less costly than maintaining a large hash table.

Second, we optimized the data structure of the polynomials in AMULET2.1. In AMULET2.0, we store the polynomials as a linked list of monomials using `std::deque`. Using deques proved to be faster than `std::vector`, but they have a higher minimal memory cost. Using an array for storing the monomials in AMULET2.1 decreased the memory usage by around 10% without affecting the verification time.

As a third modification, we changed the format of the proof certificates in AMULET2.1. In AMULET2.0, certificates can be generated in the Nullstellensatz proof format [21] or in PAC [27] to validate the verification results. For proof checking, two different proof checkers are necessary, depending on which proof calculus is selected. Recently, we developed LPAC (PAC + linear combinations) that unifies both proof formats [28]. LPAC proofs consist of a sequence of linear combination rules, cf., Example 5. It is possible to simulate Nullstellensatz by writing down only a single linear combination step, as well a PAC proofs by simulating single addition and multiplication steps. Furthermore, it is possible to generate certificates on an intermediate density level, i.e., a sequence of linear combinations. These proofs are more concise than pure PAC proofs, but still allow to localize errors. AMULET2.1 supports LPAC proofs on three density levels, which is demonstrated in the following Sect. 5.

Example 5 Let the setting be as in Example 4. A possible proof in LPAC is as follows, using the same labels for the given polynomials:

```
4 $\%$ 3*(-ab) + 2*(-y), xy;
5 $\%$ 4 + 1*(-1), z;
```

5 Usage

In this section, we demonstrate the usage of our tool AMULET2.1. The usage for AMULET2.0 is almost identical, only the options in the *certification mode* differ, cf. Sect. 5.3, due to the change in the proof formats. AMULET2.1 relies on the AIGER library [5] and the GMP library [11]. The AIGER library is provided together with the source code of AMULET2.1, the GMP library needs to be preinstalled by the user. AMULET2.1 is compiled executing "`./configure.sh && make`."

In a complete workflow, one should first apply adder substitution, using the *substitution mode*, to make sure that a potential complex FSA is replaced by a simple RC adder. Afterward, one of the two modes, the *verification mode* or *certification mode*, can be applied to verify the (simplified) multiplier, which we will call in the following *rewritten* multiplier. If it is known that the final stage adder is not a complex GP adder, the substitution step can be omitted. Figure 4 shows our tool chain used for verifying (left side) and certifying (right side) multiplier circuits. We present a complete demonstration for the unsigned 64-bit multiplier `<bpwtcl.aig>`, which is included in the complementary material [16]. The output of AMULET2.1 can be seen in the corresponding log-files that are also included in the artifact [16].

5.1 Adder substitution

First, we apply adder substitution by running

```
./amulet -substitute bpwtcl.aig miter.
    cnf rewritten.aig [options]
```

If the multiplier computes multiplication of signed integers, the option "`-signed`" has to be involved, because the signedness is part of the circuit specification.

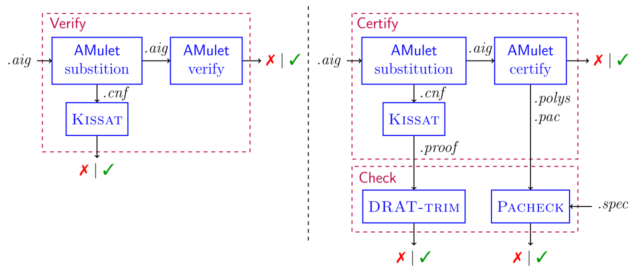

Fig. 4 Tool chain for verification (left) or certification and checking (right)

If adder substitution can be applied successfully, the generated miter is written to <miter.cnf> and the rewritten multiplier to <rewritten.aig>. Otherwise, a trivial unsatisfiable CNF is written to <miter.cnf> and the given multiplier will be written to <rewritten.aig>. The file <miter.cnf> has to be given to a SAT solver, e.g., KISSAT [4], which is then expected to return *unsatisfiable*. The rewritten multiplier can be verified or certified using AMULET2.1.

5.2 Verification

Verification is executed by

```
./amulet -verify rewritten.aig [options]
```

As for adder substitution, one has to invoke the option "-signed" for verification of signed multipliers. Furthermore, the option "-no-counter-examples" is available, which turns off generation and saving of counter examples, in the case when the circuit in <rewritten.aig> is incorrect.

5.3 Certification

Certification is applied using

```
./amulet -certify rewritten.aig out.pol
        out.prf out.spc [-pN] [options]
```

In this mode, AMULET2.1 verifies the multiplier and automatically generates proof certificates, which can be checked by corresponding proof checkers. AMULET2.1 supports the LPAC proof format on three different density levels, which can be selected using options "-p1", "-p2" or "-p3". The option "-p1" selects the most expanded proof, which generates a proof step for every single polynomial operation. Typically, for each D-reduction step, a multiplication and addition are applied, thus two proof steps are generated for each reduction step. These proof certificates are close to PAC proofs.

The option "-p2" combines the multiplication and addition steps of each D-reduction, into a single proof step. Thus, the proof files typically have only 50% of the size of "-p1"-proofs, but we are still able to associate errors in the proofs to single D-reduction steps. The option "-p2" is the default proof format in AMULET2.1.

The option "-p3" is the most concise proof format and only generates a single proof rule, i.e., a single linear combination. Proofs with option "-p3" simulate Nullstellensatz proofs. All options of the verification mode are available too.

The proof is stored in the provided files <out.pol>, <out.prf>, and <out.spc>. The file <out.pol> contains the gate constraints, the second file <out.prf> the core proof in the selected proof format and the third file <out.spc> the specification of the multiplier. The generated proofs can be given to the proof checkers PACHECK 2.0 [19], or PASTÈQUE 2.0 [10].

The involved SAT solver for checking adder equivalence produces a proof certificate in the DRUP format [12]. Since two different proof formats are involved, the generated certificates can only be trusted up to compositional reasoning. Thus, we generated a method to translate DRUP proofs into PAC [24], which can be applied on top to generate a combined proof certificate in a single proof format, see for example the experimental evaluation in [28].

6 Evaluation

In our experiments, we use an Intel Xeon E5-2620 v4 CPU at 2.10 GHz (with turbo-mode disabled) with a memory limit of 128 GB. The time is listed in seconds (wall-clock time). We compare AMULET2 to our previous tool AMULET1 and to the most recent related work RevSCA-2.0, RevSCA [34], DyPoSub [35], and ABC-based work [8] on multiplier verification using computer algebra, where circuits are given as AIGs.

The tool RevSCA and its successor RevSCA-2.0 [34] use a preprocessing technique that aims to detect converging gate cones and atomic blocks, such as half and full adders, in the given AIG. For each converging gate cone and atomic block, a vanishing-free specification polynomial is generated, i.e., a polynomial where all monomials are removed that will reduce to zero later in the reduction process. This helps to compress the Gröbner basis before reduction is applied. DyPoSub is a follow-up work of [34] and explicitly tries to tackle the problem of verifying multiplier circuits, where logic synthesis and technology mapping is applied. The main idea in DyPoSub is to use a dynamic substitution order that allows to keep the size of the intermediate reduction results on a moderate level. The preprocessing steps in DyPoSub are the same as in its predecessor RevSCA-2.0. In ABC-based work of [8], a method called *function extraction* is used 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. 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, the remainder polynomial needs to be compared to the desired circuit functionality.

In our experiments, we consider two versions of our tool AMULET1: (i) AMULET1.0 as published in [23], (ii)

AMULET1.5 a slightly improved version [15] with new heuristics for detecting GP adders. For AMULET2, we consider the version AMULET2.0 as published in [22] as well as its derivative AMULET2.1, which we discussed in Sect. 4.

For all AMULET versions, we measure and sum up the time of adder substitution, verification and the time the SAT solver KISSAT needs, i.e., all steps that are included in the rectangle on the left side of Fig. 4. All experimental data, benchmarks and the source code of AMULET2.1 are available at [20].

In our first experiment, we consider the comprehensive AOKI benchmark set [14] that provides 384 signed and unsigned integer multiplier architectures up to input bit-width 64. The AOKI benchmark set combines a wide range of PPGs, PPAs, and FSAs, including Booth encoding, Wallace tree accumulation, and carry-lookahead adders. We consider all 384 possible architectures of bit-width 64. The time limit in our experiments is set to 300 seconds. The results are shown in Figs. 5 and 6, where it can be seen that AMULET2 is the only tool that is able to verify the complete benchmark set within the given time limit. ABC-based work of [8] uses an optimization, which only works for simple multiplier architectures. Enabling this optimization on the more involved AOKI benchmarks leads to incompleteness. Without enabling it [8] either produces a segmentation fault or exceeds the time limit. Thus, there are no results for [8] in Figs. 5 and 6. It can be seen in Fig. 5 that AMULET2.0 and its derivatives are faster than the predecessor AMULET1.0 and clearly outperforms related work. Figure 6 shows the memory usage of the tools and it can be seen that all versions of AMULET use less memory than tools of related work. However, AMULET2.0 is less memory efficient than AMULET1.0, and AMULET1.5. This flaw has been fixed in AMULET2.1, which needs significantly less memory than AMULET2.0.

In our second experiment, we generate benchmarks of simple multipliers up to input size 2 048, using scripts by Arist Kojevnikov [13]. The time limit is set to 86 400 seconds (24 h) and the results are shown in Figs. 7 and 8. Since we are considering simple multiplier architectures, we can make use of the optimization in ABC-based work of [8]. It can be seen that AMULET2 outperforms all competitor tools and is an order of magnitude faster on large multiplier circuits. The memory usage is shown in Fig. 8, where it can be seen that ABC-based work of [8] is the most memory efficient.

In our third experiment, we generate large benchmarks of complex multipliers up to input size 1 024 using MULTGEN [38]. We choose the architecture "bp-wt-lf" that uses a Booth encoding to generate the partial products, which are then accumulated using a Wallace tree. The final stage adder is a Ladner-Fischer adder. The time limit is again set to 86 400 seconds (24 h), and the results are shown in Fig. 9 and 10. Since "bp-wt-lf" is a complex architecture, we had to disable the optimization in the ABC-based work, which leads to segmentation faults. The tool RevSCA exceeded the time limit

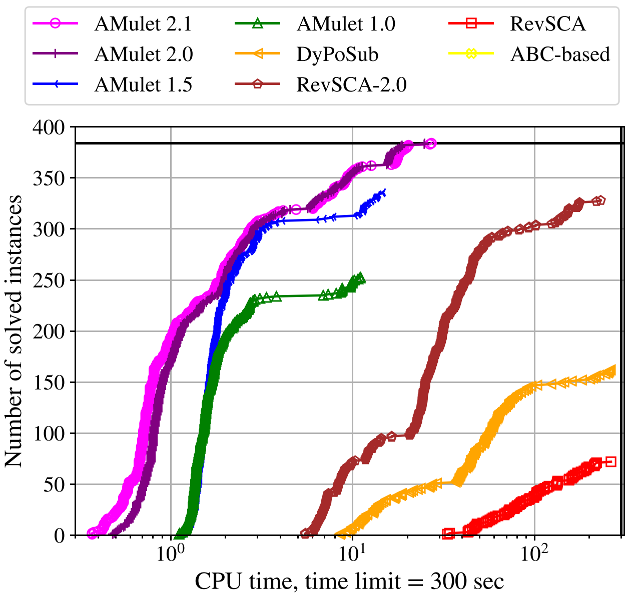

Fig. 5 Verification time of AOKI multipliers, 384 instances

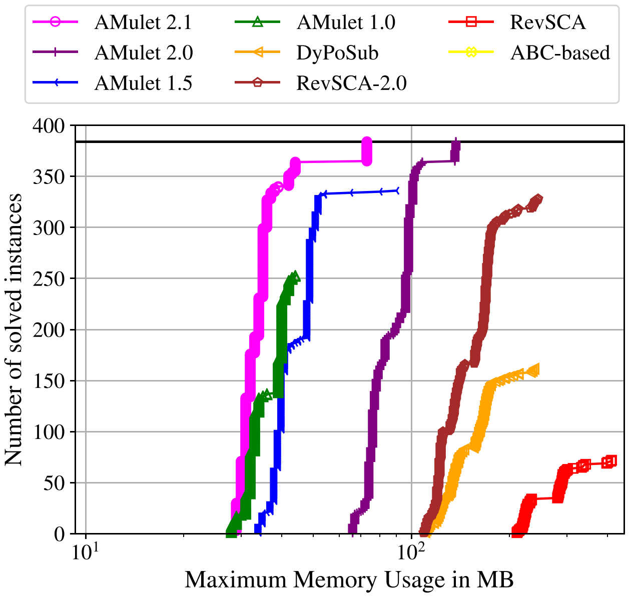

Fig. 6 Maximum memory usage of AOKI benchmark set

for all multipliers. Furthermore, an error in the XOR-based slicing approach of AMULET2.0 leads to incorrect results. This bug is fixed in later versions. Thus, we only show the results of AMULET2.1.

It can be seen in Fig. 9 that AMULET2.1 is more than an order of magnitude faster than related tools, however, AMULET2.1 requires twice as much memory than AMULET1.0 resp. AMULET1.5, cf. Fig. 10. The cause is currently unclear and investigating why AMULET2 is more memory hungry than AMULET1 on large complex multipliers is an interesting future work.

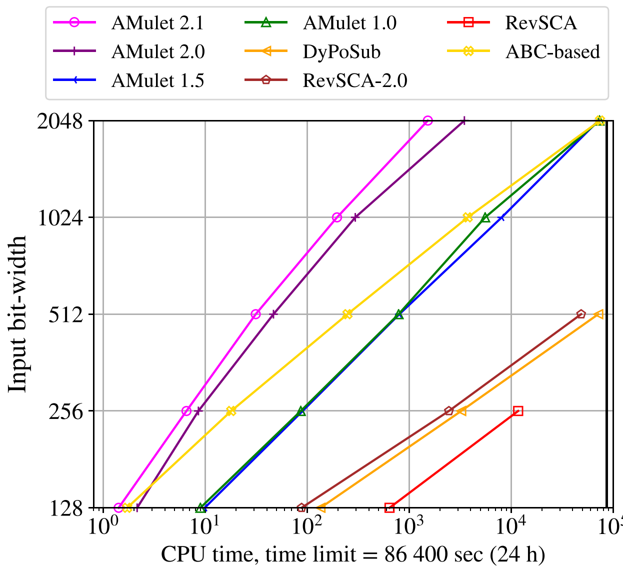

Fig. 7 Verification time of large simple multipliers

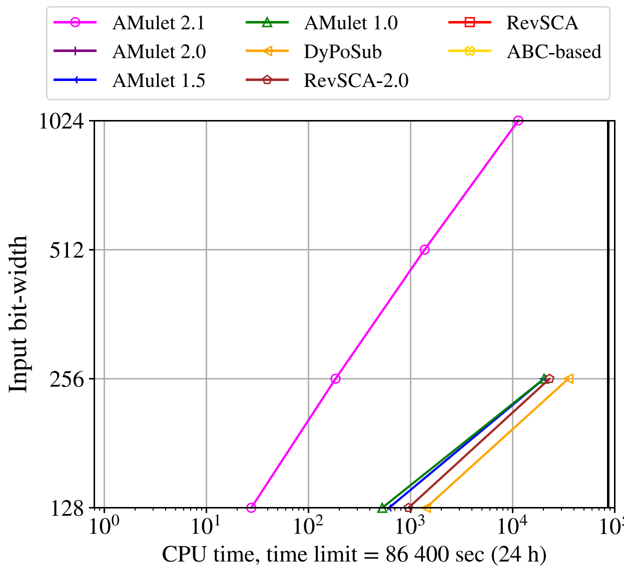

Fig. 9 Verification time of large complex multipliers

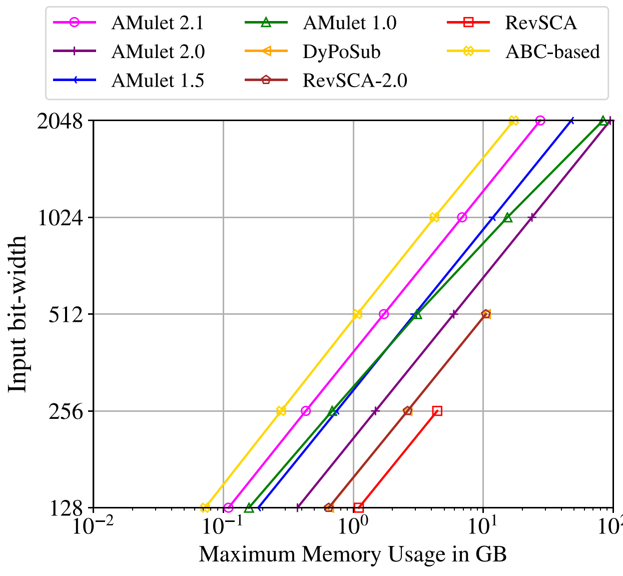

Fig. 8 Maximum memory usage of large simple multipliers

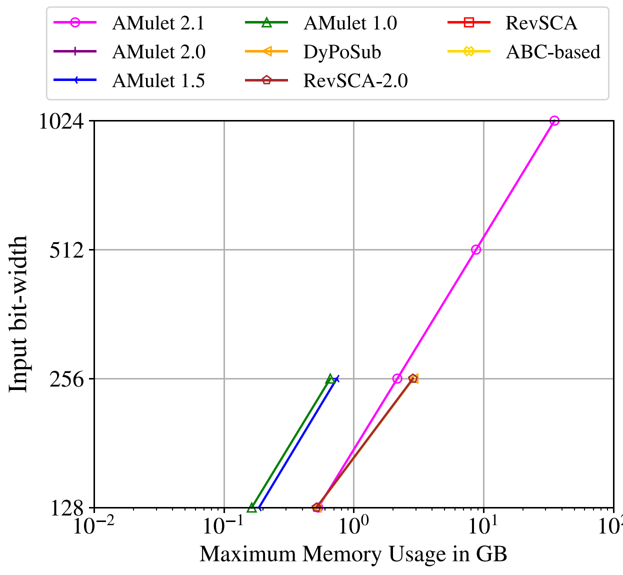

Fig. 10 Maximum memory usage of large complex multipliers

7 Conclusion

We presented AMULET2, a fully automatic tool for verifying multiplier circuits given as AIGs. AMULET2 is a re-factorization and re-implementation of our previous verification tool AMULET1 [23,26] and successfully verifies a large set of multiplier architectures. We further discussed novelties in the maintained version AMULET2.1, which help to reduce the memory usage of AMULET2.0 significantly. In the future, we want to directly integrate a SAT solver into AMULET2 and provide language bindings, e.g., for Python.

Funding Open access funding provided by TU Wien (TUW).

References

1. Beame, P., Impagliazzo, R., Krajícek, J., Pitassi, T. and Pudlák, P.: Lower bounds on Hilbert's Nullstellensatz and Propositional Proofs. In: Proc. London Math. Society, volume s3-73, pp. 1–26, (1996)
2. Becker, T., Weispfenning, V. and Kredel, H.: Gröbner Bases, volume 141 of Grad. texts in math. Springer, (1993)
3. Biere, A.: 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)
4. Biere, A., Fazekas, K., Fleury, M., and Heisinger, M.: 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)
5. Biere, A., Heljanko, K. and Wieringa, S.: AIGER 1.9 And Beyond. Technical report, FMV Reports Series, JKU Linz, Austria, (2011)
6. Bryant, R.E., Chen, Y.: Verification of arithmetic circuits using binary moment diagrams. STTT **3**(2), 137–155 (2001)
7. Buchberger, B.: Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal. PhD thesis, Univ. of Innsbruck, (1965)
8. Ciesielski, M.J., Su, T., Yasin, A., Yu, C.: Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model. IEEE TCAD **39**(6), 1346–1357 (2020)
9. Cox, D., Little, J., O'Shea, D.: Ideals, Varieties, and Algorithms. Springer-Verlag, New York (1997)
10. Fleury, M.: Isabelle PAC formalization. Theory files at https://bitbucket.org/isafol/isafol/src/master/PAC_Checker2/, Accessed: 2021-07-28
11. Granlund, T. and the GMP development team. GNU MP: The GNU Multiple Precision Arithmetic Library, 2016. Version 6.1.2
12. Heule, M.J.H., Biere, A.: Proofs for satisfiability problems. In All about Proofs, Proofs for All Workshop, APPA 2014, volume 55, pages 1–22. College Publications, (2015)
13. Hirsch, E., Itsykson, D., Kojevnikov, A., Kulikov, E. and Nikolenko, S.: Report on the Mixed Boolean-Algebraic Solver. Technical report, Laboratory of Mathematical Logic of St. Petersburg Dep. of Steklov Institute of Mathematics, (2005)
14. Homma, N., Watanabe, Y., Aoki, T., Higuchi, T.: Formal design of arithmetic circuits based on arithmetic description language. IEICE Trans. **89–A**(12), 3500–3509 (2006)
15. Kaufmann, D.: Amulet 1.5. https://github.com/d-kfmnn/amulet, (2020)
16. Kaufmann, D.: Artifact for AMulet2.0 for verifying multiplier circuits. http://fmv.jku.at/amulet2_artifact, (2020)
17. Kaufmann, D.: Formal Verification of Multiplier Circuits using Computer Algebra. PhD thesis, Informatik, Johannes Kepler University Linz, (2020)
18. Kaufmann, D.: AMulet2 for verifying multiplier circuits. https://github.com/d-kfmnn/amulet2, (2021)
19. Kaufmann, D.: Practical algebraic calculus proof checker 2.0. https://github.com/d-kfmnn/pacheck2, (2021)
20. Kaufmann, D.: Artifact for AMulet2.1. https://zenodo.org/record/6637319, (2022)
21. Kaufmann, D. and Biere, A.: Nullstellensatz-proofs for multiplier verification. In CASC, volume 12291 of LNCS, pp. 368–389. Springer, (2020)
22. Kaufmann, D. and Biere, A.: AMulet 2.0 for verifying multiplier circuits. In TACAS (2), volume 12652 of LNCS, pages 357–364. Springer, (2021)
23. Kaufmann, D., Biere, A. and Kauers, M.: Verifying large multipliers by combining SAT and computer algebra. In FMCAD 2019, pp. 28–36. IEEE, (2019)
24. Kaufmann, D., Biere, A. and Kauers, M.: From DRUP to PAC and back. In DATE 2020, pp. 654–657. IEEE (2020)
25. Kaufmann, D., Biere, A., Kauers, M.: Incremental Column-wise verification of arithmetic circuits using computer algebra. FMSD **56**(1), 22–54 (2020)
26. Kaufmann, D., Biere, A. and Kauers, M.: SAT, Computer Algebra, Multipliers. In: Vampire 2018 and Vampire 2019, volume 71 of EPiC Series in Computing, pp. 1–18. EasyChair, (2020)
27. Kaufmann, D., Fleury, M. and Biere, A.: Pacheck and Pastèque, Checking Practical Algebraic Calculus Proofs. In FMCAD 2020, volume 1 of FMCAD, pp. 264–269. TU Vienna Academic Press, (2020)
28. Kaufmann, D., Fleury, M., Biere, A. and Kauers, M.: Practical Algebraic Calculus and Nullstellensatz with the Checkers Pacheck and Pastèque and Nuss-Checker. FMSD, 2021. Submitted
29. Kaufmann, D., Kauers, M., Biere, A. and Cok, D.: 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. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.: Robust Boolean reasoning for equivalence checking and functional property verification. IEEE TCAD **21**(12), 1377–1394 (2002)
31. Lichtblau, D.: Effective computation of strong Gröbner bases over Euclidean domains. Illinois J. Math. **56**(1), 177–194 (2012)
32. Lv, J., Kalla, P., Enescu, F.: Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits. IEEE TCAD **32**(9), 1409–1420 (2013)
33. Mahzoon, A., Große, D. and Drechsler, R.: PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers. In ICCAD 2018, pp. 129:1 – 129:8. ACM, (2018)
34. Mahzoon, A., Große, D. and Drechsler, R.: RevSCA: using reverse engineering to bring light into backward rewriting for big and dirty multipliers. In DAC 2019, pp. 185:1–185:6. ACM, (2019)
35. Mahzoon, A., Große, D., Scholl, C. and Drechsler, R.: Towards formal verification of optimized and industrial multipliers. In DATE, pp. 544–549. IEEE, (2020)
36. Parhami, B.: Computer Arithmetic - Algorithms and Hardware designs. Oxford University Press, (2000)
37. Sharangpani, H. and Barton, M.L.: Statistical analysis of floating point flaw in the Pentium processor (1994)
38. Temel, M.: MultGen. https://github.com/temelmertcan/multgen, (2020)
39. Temel, M., Slobodová, A. and Hunt, W. A.: Automated and scalable verification of integer multipliers. In CAV, volume 12224 of LNCS, pp. 485–507. Springer (2020)