FastPoly: An Efficient Polynomial Package for the Verification of Integer Arithmetic Circuits

Alexander Konrad Christoph Scholl
University of Freiburg, Freiburg, Germany
{konrada, scholl}@informatik.uni-freiburg.de

Artifact ✓ Reviewed 10.5281/zenodo.16744818

***Abstract*—In recent years, methods based on Symbolic Computer Algebra (SCA) have become increasingly successful in the field of formal verification of arithmetic circuits. While several different approaches have been proposed to tackle this challenging task, most of them are based on the same mathematical operations. They perform ideal membership tests that reduce specification polynomials by a series of polynomial divisions. For integer arithmetic, under certain conditions, the polynomial divisions boil down to substitutions of variables in integer polynomials. In this context, the overall performance of an SCA verification tool is closely related to the efficiency of the steps manipulating integer polynomials. In this paper we present our tool FastPoly, a package for representing integer polynomials that provides efficient operations including variable substitution with integrated normalization steps. We provide the sources of our tool, making it available for other research groups to support future progress in this field.**

I. Introduction

Arithmetic circuits play an important role in circuit designs, ranging from general-purpose processors to specialized hardware used in computationally intensive applications such as cryptography or machine learning. In order to increase the confidence in the designs and to avoid design errors such as the infamous Pentium bug [1], fully automatic formal verification became more and more important.

In particular, the verification of multiplier and divider circuits has been a challenging problem for a long time. Methods based on BDDs [2], [3] struggle with exponential space complexity while SAT-based methods [4], [5] experience exponential run times. However, methods based on *Symbolic Computer Algebra* (SCA) have become increasingly successful in recent years, enabling the verification of large and complex arithmetic circuits like finite field multipliers [6], integer multipliers [7]–[23], modular multipliers [24] and divider circuits [25]–[29]. Here the verification task has been reduced to an ideal membership test for the specification polynomial based on so-called backward rewriting, sometimes also referred to as algebraic reasoning [19], [20], [22], [30].

In the context of SCA-based formal verification for integer arithmetic, a variety of different approaches have been published recently. Some focus on reverse engineering and detection of converging cones to precompute polynomials for sub-circuits and simplify those polynomials early on [15], [16], [18], [21]. Some rely on adder detection to simplify the circuit under verification and additionally split the ideal membership test into several sub-tasks by splitting the specification into "slices", one for each primary output [17], [19], [20], [22], [30]. Others precompute specific signal relations in the circuit like equivalences/antivalences [27] or satisfiability don't cares [28], [29] to simplify occurring intermediate polynomials which especially is necessary for divider verification. One of the most recent methods showed that a successful verification of integer multipliers can be achieved by only using dynamic phase and order optimization [23].

Despite differences in the way their algorithms work, at the lowest level all of these approaches rely on the same basic operations: Representing specification polynomials and performing manipulation steps on them, either by polynomial divisions or the simpler substitutions of variables by integer polynomials. In the remainder of the paper we will refer to those manipulations as *substitution steps*. Those substitution steps influence the higher-level algorithms in several ways: First, the overall performance of the algorithm is closely related to the efficiency of single substitution steps. This is easy to conclude, because each approach ultimately boils down to a certain number of necessary substitution steps that it must perform. Second, the efficiency of single substitution steps already influences the design options of the algorithms. If a single substitution step already has high computational costs, algorithmic approaches reducing the number of substitution steps or splitting the specification polynomial into smaller slices become more important and elaborate methods searching for good substitution orders to reduce peak sizes of polynomials as used in [23] become infeasible.

In this paper we present FastPoly, an integer polynomial package that provides efficient operations on polynomials including the substitution steps which are the basis for many SCA-based algorithms. We elaborate how we achieve an efficient implementation of those substitution steps by using two simple data structures. Our experimental results show the efficiency of our package in contrast to other publicly available polynomial packages which provide a similar operation set. By providing the sources of our package and making them available for other research groups, we encourage and support future progress in this field.

The paper is structured as follows: In Sect. II we provide details on our polynomial package. In Sect. III we give a demonstration of how to use our package. We evaluate our

https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_20

This article is licensed under a Creative Commons Attribution 4.0 International License

package in Sect. IV and conclude the paper with final remarks in Sect. V.

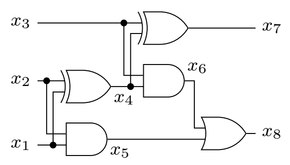
II. Tool Description

A. Requirements

Before diving into the details of our polynomial package, we first summarize the requirements for a polynomial package that is suitable to be used in SCA-based approaches to the verification of integer arithmetic circuits.

We start by defining the polynomials we want to represent as well as their basic characteristics: For integer arithmetic we consider polynomials over binary variables (from a set $X = \{x_1, \ldots, x_n\}$) with integer coefficients from $\mathbb{Z}$, i. e., a polynomial is a sum of monomials, a monomial is a product of a term with an integer, and a term is a product of variables from X. Polynomials represent *pseudo-Boolean functions* $f : \{0,1\}^n \mapsto \mathbb{Z}$. The very basic operation that our polynomial package has to provide (in an efficient way) is the substitution step of the SCA-based backward rewriting approach. In this step all occurrences of a distinct variable x_i in the polynomial are replaced by some other polynomial P_i, i. e., the substitution with $x_i = P_i$ (or the polynomial division with the polynomial $-x_i + P_i$) is performed. To achieve canonical representations, polynomials also have to be simplified after each substitution step by reducing powers v^k of variables v with $k > 1$ to v (since the variables are binary), by combining monomials with identical terms into one monomial, and by omitting monomials with leading factor 0. For those normalization steps we need the functionalities of finding, adding and removing monomials. For example, if we want to add a monomial $c \cdot m$ (which originates from a substitution step or has to be added for other reasons) to our polynomial, we first have to search for a monomial $d \cdot m$ that already exists in the polynomial. In this case, the old $d \cdot m$ is removed and the new monomial $(c + d) \cdot m$ (if $c + d \neq 0$) is added to the polynomial. In this way we make sure that there is only up to one monomial with the same term in the polynomial. In the context of circuit verification, x_i typically represents the output of a circuit component and P_i is the polynomial description of the function of this component expressed in its input variables. Such components can be basic logic gates such as AND and OR gates with the corresponding polynomials being $P_i = a_i \cdot b_i$ and $P_i = a_i + b_i - a_i \cdot b_i$, respectively, but they can also describe larger parts of the circuit, such as fanout-free cones, as used in [16]. Fig. 1 shows the series of substitution steps for a full adder circuit.

In summary, our polynomial package has to represent and normalize polynomials as described above. We need to be able to find monomials in the polynomial quickly, we have to add and remove monomials quickly, and we need an efficient way of carrying out the basic substitution step. Next we describe how we achieve those goals in our polynomial package FastPoly.



$$\begin{aligned}
x_8 &= x_5 + x_6 - x_5x_6\\
x_7 &= x_3 + x_4 - 2x_3x_4\\
x_6 &= x_3x_4\\
x_5 &= x_1x_2\\
x_4 &= x_1 + x_2 - 2x_1x_2
\end{aligned}$$

$$\begin{aligned}
& 2x_8 + x_7\\
\xrightarrow{x_8}\ & 2x_5 + 2x_6 - 2x_5x_6 + x_7\\
\xrightarrow{x_6}\ & 2x_5 + 2x_3x_4 - 2x_3x_4x_5 + x_7\\
\xrightarrow{x_7}\ & 2x_5 - 2x_3x_4x_5 + x_3 + x_4\\
\xrightarrow{x_5}\ & 2x_1x_2 - 2x_1x_2x_3x_4 + x_3 + x_4\\
\xrightarrow{x_4}\ & x_1 + x_2 + x_3
\end{aligned}$$

Fig. 1. Full adder circuit with series of substitutions.

B. Implementation

Our polynomial package FastPoly is implemented in C++. In this subsection we break down the data structures used and how they support the necessary operations.

1) Polynomials: Our class for polynomials, called *Polynom*, is the data structure saving all information needed for the representation of integer polynomials and enabling an efficient implementation of the substitution operation. The class consists of two major parts. The first part uses the std::set from C++ for saving all monomials of a polynomial. We elaborate on the monomial class in the next subsection. For now it is only important to know that we have monomial objects representing monomials and those monomials have a specific order defined among each other which is used to determine a monomial's position in the set. The std::set is usually implemented as red-black tree. Therefore, searching, inserting and removing monomials have logarithmic time complexity in the size of the set, i. e., the size of the polynomial.

However, for the desired substitution operation, where we substitute all occurrences of a variable x_i in the polynomial by some other polynomial P_i, this set is not sufficient to enable an efficient implementation of the operation. As mentioned, searching for a monomial has logarithmic time complexity, which is good enough for our needs as long as we are searching for a single specific monomial. In contrast, for the substitution operation we do not search for a single monomial only, but we have to find all monomials which include the variable x_i we want to substitute. To avoid the necessity of searching through the complete polynomial for this purpose, we use a second index data structure for our polynomial class which we call *refList*. Basically, we maintain for every variable in the polynomial a doubly linked list. The elements stored in the doubly linked list for some variable x_i are pointers to all monomials containing x_i. The heads of the doubly linked lists are stored in a C-style array. Now, for the substitution operation we find all monomials containing a specific variable x_i by obtaining (in constant time) the head of the list for x_i from the array under index i and by following the doubly linked list afterwards. If a variable is not present in the polynomial, the corresponding list will be empty, which can be checked in constant time as well.

2) Monomials: Now, we define the implementation of monomials in detail. Our class for monomials is called *Monom* and consists of the following: The coefficient is realized by an arbitrary-precision integer implemented by the GNU Multiple Precision Arithmetic Library (GMP) [31]. A usual 64-bit integer is not sufficient here since in circuit verification we typically deal with very large coefficients. The term of the monomial is a C-style array of integers which is always sorted in ascending order. An integer i represents the variable x_i. We use an additional integer to save the size of the array and an additional integer to save the sum over all indices of variables in the monomial. This sum can be seen as a very light-weight form of a hash value for each monomial which is used to differentiate between monomials more quickly. For saving the monomials into the std::set of the polynomial class, we have to define a specific order on monomials. To compare two monomials, we first compare the sum components of the monomials. In case the sums are identical, we compare the sizes of the monomials next. If the sizes are identical as well, we fall back to a lexicographic order of the terms of the compared monomials. For this lexicographic order, we may need, in the worst case, to compare all variables in both terms of the compared monomials. For this reason we compare the sum and size attributes of monomials first, since most monomials differ already either in their sum or their size. Note that the coefficients of monomials are *not* used for ordering. There is only up to one monomial with the same term in a polynomial and during the addition of a monomial $c \cdot m$ to a polynomial p we just look for *some* existing monomial $d \cdot m$ in p with arbitrary coefficient d. Additionally, the representation of a monomial m contains for each variable x_i a "back" pointer to the element in the doubly linked list for x_i that points to m. In that way, m can be removed from the index data structure formed by the doubly linked lists in time that is linear in the size of m. Fig. 2 provides a visualization of the explained data structures for the simple example polynomial $P = 5x_1 + 7x_2 + 9x_1x_3$.

C. Additional Features

In addition to the basic structure of our polynomial package explained in the previous subsection, we provide the following features:

- Internal modulo reduction: Our package provides the possibility to set a modulo reduction parameter, enabling an internal modulo operation on monomial coefficients already during internal polynomial operations. This ensures that all monomial coefficients are calculated modulo the reduction parameter, which is used in many SCA-based approaches [23].
- Phase Optimization: We offer the functionality to change the "phase" of a variable in the polynomial by replacing each occurrence of the variable by its negation. This can be used to perform phase optimization with the goal to reduce the polynomial size [23].
- Certification: We offer the ability to produce certificates in the practical algebraic calculus (PAC) format [32].

III. Tool Usage

We provide the source code of our tool at [33]. Our tool relies on the GMP library [31] which has to be preinstalled by the user. We currently provide three different APIs for the usage of our polynomial package (which also can be used in a mixed fashion):

1) Reading in the starting polynomial and the polynomials for the substitution steps from an external file.
2) Building polynomials from scratch using the constructors of the polynomial and monomial classes and performing substitution steps of variables by polynomials by the basic substitution function.
3) Using pre-defined "shortcut" functions for substitution steps of basic logic gates.

In Listing 1 we provide a demonstration on how to use these APIs for the simple example of backward rewriting a full adder circuit which is shown in Fig. 1.

IV. Experimental Results

We have tested the performance of our package on a number of sample cases. The tests were executed on a single core of an Intel Xeon CPU E5-2643 with 3.30 GHz. Resources were limited to 32 GB of main memory and 6 hours of CPU time. For comparison we ran the same experiments for the general-purpose computer algebra system SINGULAR [34] as well as for the polynomial library used in AMULET2.2 [30], since it is, to the best of our knowledge, the only other publicly available polynomial package for the specific purpose of SCA-based verification.

We provide an artifact at [35] for better reproducibility of our experimental results.

For the benchmark set we decided to use some examples from SCA-based multiplier verification [23]. Since we want to evaluate the performance of our presented package and compare it to the polynomial library from AMULET2.2 and to the computer algebra system SINGULAR, we abstract from differences in the application of the polynomial packages in our verification tool DYNPHASEORDEROPT [23] and in AMULET2.2. We achieve this by using benchmarks which consist only of a starting polynomial (which is equal to the specification polynomial from [23]) and a (pre-computed) series of substitution steps. To create these benchmarks, we use DYNPHASEORDEROPT which intensively searches for a "good" traversal order (in terms of small intermediate polynomial sizes) of the gates of a given multiplier circuit. It achieves this by determining candidates for the next substitutions, trying out these substitution steps and, if the polynomial size grows too large, reverting the steps again until a suitable series of substitutions is found. We write out the resulting substitution order from DYNPHASEORDEROPT to be able to "replay" it later on. In summary, the experiments we consider here consist only of construction and substitution of polynomials, with the substitution steps predetermined in the benchmark files. Note that DYNPHASEORDEROPT actually performs an additional "phase optimization" after every step. The phase

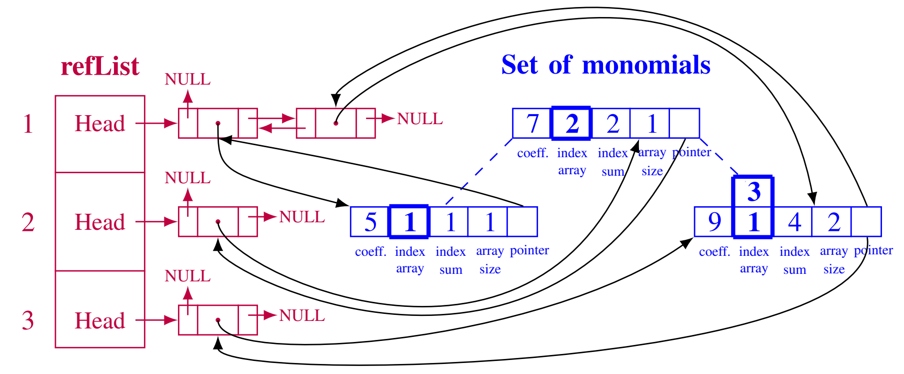


Fig. 2. Visualization of our polynomial data structures.

TABLE I
EXPERIMENTAL RESULTS. TIMES IN CPU SECONDS.

Benchmark	# steps	Starting poly.size	Run times FastPoly	AMULET2.2	Singular	Maximum poly.size
16-bit sp-ar-rc	2,816	288	0.85	0.82	TO	302
32-bit sp-ar-rc	11,776	1,088	2.53	9.08	TO	1,102
64-bit sp-ar-rc	48,128	4,224	9.69	103.58	EE	4,298
128-bit sp-ar-rc	194,560	16,640	38.77	7444.54	EE	16,654
256-bit sp-ar-rc	782,336	66,048	150.35	TO	EE	66,062
16-bit sp-wt-lf	3,057	288	0.58	1.24	TO	670
32-bit sp-wt-lf	12,616	1,088	2.41	20.97	TO	2,055
64-bit sp-wt-lf	50,564	4,224	9.84	455.97	EE	8,315
128-bit sp-wt-lf	200,100	16,640	38.65	TO	EE	26,522
256-bit sp-wt-lf	796,191	66,048	158.00	TO	EE	82,154
8-bit sp-ar-ks	652	80	1.77	TO	TO	54,473

optimization is omitted, since the polynomial library from AMULET2.2 does not support this operation. We considered three different multiplier architectures: sp-ar-rc, sp-wt-lf, and sp-ar-ks [36]. Usually, integer multipliers are composed of three stages: The first stage is the Partial Product Generator (PPG) which generates partial products from the bits of the two input operands. In our considered architectures this is realized by a simple PPG (sp), which just computes the logical AND of all bits of the first input and all bits of the second input. The second stage is the Partial Product Accumulator (PPA) which sums up all the partial products until they are reduced to two numbers only. In our experiments we consider array accumulation (ar) and Wallace trees (wt) [37]. The third stage consists of the Final Stage Adder (FSA) which converts the resulting two numbers from the PPA stage into the final binary representation of the output product, realized by a two operand adder network. We consider the well-known ripple-carry adder (rc), the Ladner-Fischer adder (lf) [38] and the Kogge-Stone adder (ks) [39]. The former two architectures sp-ar-rc and sp-wt-lf can be considered as easy to verify and lead to rather low peak sizes of the polynomials when choosing a reasonable substitution order. The latter, sp-ar-ks, is more challenging for SCA-based verification tools. Therefore, we consider bitwidths ranging from 16 to 256 bits for the two easy architectures, while we only consider the 8-bit benchmark for sp-ar-ks. We would like to point out that we have deliberately chosen a few architectures as examples for which we know that the computed series of substitution steps does not produce excessively large intermediate polynomials, even without phase optimization. A comprehensive experimental analysis showing the robustness of our verification tool DYNPHASEORDEROPT on different multiplier architectures can be found in [23]. For example, the successful verification of challenging circuits like the sp-ar-ks multipliers for much larger bitwidths than eight (as considered here) definitely needs the dynamic phase optimization from [23] as well.

The experimental results are shown in Tab. I. Col. 1 states the benchmark. Col. 2 gives the number of substitution steps for this benchmark and Col. 3 the size of the starting polynomial. Col. 4 indicates the run time for our package to execute all substitution steps while Col. 5 gives the run time for the polynomial library from AMULET2.2 [30]. Col. 6 shows the results for the computer algebra system Singular. Col. 7 shows the maximum polynomial size reached by intermediate results for our package as well as for AMULET2.2. Since we perform the same series of substitutions for both packages, they produce the same intermediate results leading to identical maximum polynomial sizes. However, we cannot confirm this for SINGULAR, because it does not provide intermediate results.

Listing 1. Full adder demo.

```
#include "poly_parser.h"

int main(int argc, char ** argv) {

// Example 1) Reading in from external file.
Polynom spec1;
init_spec(spec1, "./demo/fulladder_example.txt");
reduce_poly(spec1, "./demo/fulladder_example.txt");
std::cout << "Result1:" << spec1 << std::endl;

//Example 2) Building polynomials from scratch.
Polynom x8poly(6); // Instantiate poly for x8 gate.
// The initialization parameter defines the maximum
// variable index for the polynomial.
x8poly.createMonom(5); // Create and add monom 1*x5.
x8poly.createMonom(6); // Create and add monom 1*x6.
x8poly.createMonom(5,6, mpz_class(-1));
// Create and add monom -1*x5*x6.
Polynom x7poly(4); // Instantiate poly for x7 gate.
x7poly.createMonom(3);
x7poly.createMonom(4);
x7poly.createMonom(3,4, mpz_class(-2));
Polynom x6poly(4); // Instantiate poly for x6 gate.
x6poly.createMonom(3,4);
Polynom x5poly(2); // Instantiate poly for x5 gate.
x5poly.createMonom(1,2);
Polynom x4poly(2); // Instantiate poly for x4 gate.
x4poly.createMonom(1);
x4poly.createMonom(2);
x4poly.createMonom(1,2, mpz_class(-2));
Polynom spec2(8); // Instantiate spec2 polynomial.
spec2.createMonom(8, mpz_class(2));
spec2.createMonom(7);
// Now substitute xi by its gate polynomial xipoly.
spec2.replaceVarByPoly(8, x8poly);
spec2.replaceVarByPoly(6, x6poly);
spec2.replaceVarByPoly(7, x7poly);
spec2.replaceVarByPoly(5, x5poly);
spec2.replaceVarByPoly(4, x4poly);
std::cout << "Result2:" << spec2 << std::endl;

//Example 3) Using "shortcut" logic gate functions.
Polynom spec3(8); // Instantiate spec3 polynomial.
spec3.createMonom(8, mpz_class(2));
spec3.createMonom(7);
spec3.replaceOR(8, 5, 6);
// Function for replacing x8 by the OR gate
// polynomial with inputs x5 and x6.
spec3.replaceAND(6, 3, 4);
spec3.replaceXOR(7, 3, 4);
spec3.replaceAND(5, 1, 2);
spec3.replaceXOR(4, 1, 2);
std::cout << "Result3:" << spec3 << std::endl;

return 0;
}
```

It can be seen that our package clearly outperforms the polynomial library from AMULET2.2 and SINGULAR. This can be explained by the fact that in the polynomial library from AMULET2.2 the performance of a single substitution step clearly depends on the size of the current polynomial, even if only a fraction of the polynomial is actually changed. This resulted in long run times and even time outs (indicated by TO) for benchmarks which produce large intermediate polynomials like the 128-bit and 256-bit examples as well as the challenging 8-bit sp-ar-ks benchmark. SINGULAR was not able to solve any of the benchmarks. The smaller benchmarks ran into time outs (TO), while benchmarks with bitwidths of 64 and larger produced an error state (indicated by EE) because SINGULAR cannot handle more than 32767 ring variables. This result was to be expected. Although SINGULAR, as general-purpose computer algebra system, provides the necessary operations to perform the polynomial substitution steps, it is not specifically tailored for our considered use case, unlike our polynomial package as well as the one from AMULET2.2. Our package, on the other hand, was able to solve every benchmark within a few minutes. We can observe that the run times of our package increase roughly linearly with the number of substitution steps (which in turn grow quadratically with the bitwidth of the benchmark). We would like to emphasize that the efficiency of the substitution steps shown for our polynomial package is key to enabling the intensive order calculations which are performed in our verification tool from [23].

V. CONCLUSIONS AND FUTURE WORK

We have presented our tool FastPoly, a polynomial package designed for supporting efficient polynomial operations which are used in SCA-based approaches for circuit verification. Our experimental results confirm the efficiency of our tool. We believe by making our tool available for other research groups we will support future advances in the field of SCA-based circuit verification.

REFERENCES

[1] T. Coe, "Inside the Pentium FDIV bug," *Dr. Dobbs J.*, vol. 20, no. 4, pp. 129—135, 1995.

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

[3] J. R. Burch, "Using BDDs to verify multipliers," in *DAC*, 1991, pp. 408–412.

[4] J. P. M. Silva and T. Glass, "Combinational equivalence checking using satisfiability and recursive learning," in *DATE*. IEEE Computer Society / ACM, 1999, pp. 145–149.

[5] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT for combinational equivalence checking," in *DATE*. IEEE Computer Society, 2001, pp. 114–121.

[6] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits," *TCAD*, vol. 32, no. 9, pp. 1409–1420, 2013.

[7] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G. Greuel, "An algebraic approach for proving data correctness in arithmetic data paths," in *CAV*, 2008, pp. 473–486.

[8] F. Farahmandi and B. Alizadeh, "Gröbner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," *MICPRO*, vol. 39, no. 2, pp. 83–96, 2015.

[9] M. Ciesielski, C. Yu, D. Liu, and W. Brown, "Verification of gate-level arithmetic circuits by function extraction," in *DAC*, 2015, pp. 52:1–52:6.

[10] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *TCAD*, vol. 35, no. 12, pp. 2131–2142, 2016.

[11] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal verification of integer multipliers by combining Gröbner basis with logic reduction," in *DATE*, 2016, pp. 1048–1053.

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

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

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

[15] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers," in *ICCAD*, 2018, pp. 129:1–129:8.

[16] ——, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *DAC*, 2019, pp. 185:1–185:6.

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

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

[19] D. Kaufmann and A. Biere, “AMulet 2.0 for verifying multiplier circuits,” in *TACAS*. Springer, 2021, pp. 357–364.

[20] D. Kaufmann, P. Beame, A. Biere, and J. Nordström, “Adding dual variables to algebraic reasoning for gate-level multiplier verification,” in *DATE*. IEEE, 2022.

[21] A. Mahzoon, D. Große, and R. Drechsler, “RevSCA-2.0: Sca-based formal verification of nontrivial multipliers using reverse engineering and local vanishing removal,” *TCAD*, vol. 41, no. 5, pp. 1573–1586, 2022.

[22] D. Kaufmann and A. Biere, “Improving AMulet2 for verifying multiplier circuits using SAT solving and computer algebra,” *STTT*, vol. 25, no. 2, pp. 133–144, 2023.

[23] A. Konrad and C. Scholl, “Symbolic computer algebra for multipliers revisited - it’s all about orders and phases,” in *FMCAD*. IEEE, 2024, pp. 261–271.

[24] A. Mahzoon, D. Große, C. Scholl, A. Konrad, and R. Drechsler, “Formal verification of modular multipliers using symbolic computer algebra and boolean satisfiability,” in *DAC*, 2022.

[25] A. Yasin, T. Su, S. Pillement, and M. J. Ciesielski, “Formal verification of integer dividers: Division by a constant,” in *ISVLSI*, 2019, pp. 76–81.

[26] ——, “Functional verification of hardware dividers using algebraic model,” in *VLSI-SoC*, 2019, pp. 257–262.

[27] C. Scholl and A. Konrad, “Symbolic computer algebra and SAT based information forwarding for fully automatic divider verification,” in *DAC*, 2020.

[28] C. Scholl, A. Konrad, A. Mahzoon, D. Große, and R. Drechsler, “Verifying dividers using symbolic computer algebra and don’t care optimization,” in *DATE*. IEEE, 2021, pp. 1110–1115.

[29] A. Konrad, C. Scholl, A. Mahzoon, D. Große, and R. Drechsler, “Divider verification using symbolic computer algebra and delayed don’t care optimization,” in *FMCAD*. IEEE, 2022, pp. 1–10.

[30] D. Kaufmann and A. Biere, “Fuzzing and delta debugging and-inverter graph verification tools,” in *TAP@STAF*. Springer, 2022, pp. 69–88.

[31] T. Granlund and the GMP development team, “GNU MP: The GNU Multiple Precision Arithmetic Library,” 2023, https://gmplib.org/.

[32] D. Kaufmann, M. Fleury, and A. Biere, “The proof checkers pacheck and pastèque for the practical algebraic calculus,” in *FMCAD*. IEEE, 2020, pp. 264–269.

[33] A. Konrad and C. Scholl, “FastPoly source code,” 2025. [Online]. Available: https://github.com/a-konrad/fastpoly

[34] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann, “SINGULAR 4-2-1 — A computer algebra system for polynomial computations,” https://www.singular.uni-kl.de, 2021.

[35] A. Konrad and C. Scholl, “Artifact for FastPoly: An efficient polynomial package for the verification of integer arithmetic circuits,” 2025. [Online]. Available: https://doi.org/10.5281/zenodo.16744818

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

[37] C. S. Wallace, “A suggestion for a fast multiplier,” *IEEE Trans. on Electronic Comp.*, vol. EC-13, pp. 14–17, 1964.

[38] R. E. Ladner and M. J. Fischer, “Parallel prefix computation,” *Journal of the ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[39] P. M. Kogge and H. S. Stone, “A parallel algorithm for the efficient solution of a general class of recurrence equations,” *IEEE Computer*, vol. 22, no. 8, pp. 786–793, 1973.