

Eingereicht von
Dipl.-Ing.
Daniela Kaufmann, Bsc

Angefertigt am
Institut für
Formale Modelle
und Verifikation

Erstbeurteiler
Univ.-Prof. Dr.
Armin Biere

Zweitbeurteiler
Prof.
Jakob Nordström

Mitbetreuung
Univ.-Prof. Dr.
Manuel Kauers

März 2020

Formal Verification of Multiplier Circuits using Computer Algebra



Dissertation

zur Erlangung des akademischen Grades

Doktorin der Technischen Wissenschaften

im Doktoratsstudium der

Technischen Wissenschaften

Abstract

Digital circuits are extensively used in computers and digital systems because they are able to represent models for various digital components and arithmetic operations. A subclass of digital circuits are arithmetic circuits, which are used in computer circuits to perform Boolean algebra. 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 gate-level multipliers, impose a challenge for existing verification techniques and in practice still require substantial manual effort. Approaches based on satisfiability checking (SAT) or on decision diagrams seem to be unable to solve this problem in a reasonable amount of time. In principle, theorem provers in combination with SAT are able to verify industrial multipliers, but this approach cannot be applied fully automated. Currently, the most effective automated reasoning technique relies on computer algebra. In this approach the word-level specification, modeled as a polynomial, is reduced by a Gröbner basis, which is implied by the gate-level representation of the circuit. The reduction returns zero if and only if the circuit is correct.

In this thesis we give a rigorous formalization of this reasoning method including soundness and completeness arguments, first for polynomial rings, where the coefficient domain is a field and later for more general polynomial rings. As a consequence we are able to verify not only large unsigned and signed integer multipliers very efficiently, but are also able to verify truncated multipliers. We further improve the algebraic verification approach and present a new incremental column-wise verification algorithm, which splits the verification problem into smaller more manageable sub-problems and thus does not require to consider a full word-level specification. We present preprocessing approaches based on variable elimination in order to rewrite and hence simplify the implied Gröbner basis. However, certain parts of a multiplier, namely final-stage adders, are hard to verify using computer algebra. In our approach we use SAT to replace complex adders by equivalent adders, which can be verified using computer algebra. We develop a dedicated reduction engine, which is able to apply adder substitution and verifies large multipliers of input bit-width 2048 fully automated.

Nonetheless, the verification process might not be error-free. Generating and automatically checking proofs independently increases confidence in the results of automated reasoning tools. We show how the polynomial calculus can be instantiated to yield a practical algebraic calculus (PAC). Proofs in this format can be obtained as a by-product of verifying multiplier circuits in our reduction engine and can be checked with our independent proof checking tools.

Zusammenfassung

Digitale Schaltungen modellieren digitale Komponenten und arithmetische Operationen und sind daher ein essenzieller Bestandteil in Computern und digitalen Systemen. Arithmetische Schaltungen sind ein Spezialfall von digitalen Schaltungen und werden in Computern genutzt um Boole'sche Algebra zu implementieren. Es ist äußerst wichtig, dass diese Schaltungen korrekt sind, um Fehler wie zum Beispiel den berühmten Pentium-FDIV-Bug zu vermeiden.

Mithilfe von formaler Verifikation kann man feststellen, ob eine gegebene Schaltung ihrer Spezifikation entspricht. Jedoch sind arithmetische Schaltungen, insbesondere Multiplizierer auf Gatterebene, eine Herausforderung für bestehende Verifikationstechniken. Techniken basierend auf dem Entscheidungsproblem der Aussagenlogik (SAT) oder auf Entscheidungsdiagrammen sind nicht der Lage Multiplizierer effizient zu verifizieren. Theorem Prover in Kombination mit SAT können die Korrektheit komplexer Multiplizierer beweisen, allerdings ist diese Methode nicht vollautomatisch anwendbar.

Die zurzeit erfolgreichste Beweistechnik basiert auf Computeralgebra. In dieser Methode wird die Schaltung mithilfe von Polynomen als eine Gröbner Basis modelliert. Die Spezifikation, ebenfalls als Polynom kodiert, wird mittels der Gröbner Basis reduziert. Diese Reduktion liefert das Ergebnis null genau dann, wenn die Schaltung ihrer Spezifikation entspricht.

In dieser Thesis wird dieses Problem einfach, aber präzise formalisiert und Korrektheit und Vollständigkeit wird zuerst für Polynomringe über Körper und nachfolgend für allgemeinere Ringe bewiesen. Dadurch sind wir in der Lage nicht nur vorzeichenlose und -behaftete Integer-Multiplizierer, sondern auch abgeschnittene Multiplizierer zu verifizieren. Wir verbessern diese algebraische Beweistechnik und präsentieren einen neuen inkrementellen Algorithmus, der es erlaubt, das Verifikationsproblem in kleinere Teilprobleme aufzuspalten. Weiters präsentieren wir Vorverarbeitungstechniken, welche Variablen von der Gröbner Basis eliminieren und daher die Polynomdarstellung der Schaltung vereinfachen. Jedoch können gewisse Bestandteile von Multiplizierern, genauer gesagt Addierer, nicht sehr effizient mit Computeralgebra verifiziert werden. Wir nutzen SAT, um diese komplexen Addierer mit einfachen äquivalenten Addierern zu ersetzen. Wir implementieren ein dediziertes Verifikationstool und können Multiplizierer mit Bitbreite 2048 vollautomatisch verifizieren.

Nichtsdestotrotz kann das Verifikationsprogramm Fehler enthalten. Daher werden Beweiszertifikate generiert, welche von eigenständigen Beweischeckern auf Richtigkeit überprüft werden. Wir instanziierten den abstrakten "Polynomial Calculus" und formalisieren das Beweiskalkül PAC. PAC Beweise können in unserem Verifikationstool erzeugt und mit unseren eigenständigen Beweischeckern überprüft werden.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Acknowledgements

*“Great things are never done by one person.
They’re done by a team of people.”*

–Steve Jobs

This thesis would not have been possible without the help and support of many people.

First and foremost, I would like to thank my supervisor Armin Biere for his great and constant support at every phase during my PhD. Armin, thank you for always encouraging me, providing feedback, and most importantly for letting me follow my own research interests.

I am truly grateful to have you as my advisor.

Furthermore, I would like to thank my co-author Manuel Kauers who always has an open ear when it comes to algebra. I really enjoyed our fruitful meetings and discussions. Thank you for supporting me all along the way.

I am thankful to Jakob Nordström for the time and effort to evaluate this work.

I want to thank my colleagues. It was a pleasure to work (and have coffee) with you. A very special gratitude goes to the older PhD generation from our institute, who were great mentors and from whom I learned a lot what it means to be a PhD student.

I would also like to thank my family and friends for their continuous support and patience, not only in the last couple of years but during my whole life.

Last but not least, the biggest thank you goes to the most important person in my life. It’s dangerous to go alone – thank you for being part of my life and thank you for your unrestricted courage, wisdom, and power.

Contents

I	Prologue	1
1	Introduction	3
1.1	Outline	6
1.2	Contributions	6
2	Background	9
2.1	Multiplier Circuits	9
2.2	Algebra	11
2.3	Circuit Verification using Computer Algebra	12
2.4	SAT	13
2.5	Algebraic Proof Systems	14
2.6	Related Work	15
3	Paper A: Incremental Column-Wise Verification of Arithmetic Circuits Using Computer Algebra	17
3.1	Polynomial Ring	18
3.2	Incremental Algorithm	20
3.3	Reduction Ordering	21
3.4	Computer Algebra Systems	22
4	Paper B: A Practical Polynomial Calculus for Arithmetic Circuit Verification	27
4.1	Comparison to Polynomial Calculus	28
4.2	Boolean Value Constraints	28
4.3	Computer Algebra Systems	32
5	Paper C: Verifying Large Multipliers by Combining SAT and Computer Algebra	35
5.1	Variable Elimination	36
5.2	Proof Certificates	38
6	Paper D: SAT, Computer Algebra, Multipliers	41
6.1	Proof Generation	42
6.2	Proof Size	44
7	Paper E: From DRUP to PAC and Back	45
7.1	DRUP to PAC	46

7.2	PAC to DRUP	48
8	Paper F: The Proof Checkers Pacheck and Pastèque for the Practical Algebraic Calculus	51
8.1	Extensions	52
8.2	PACHECK	53
9	Evaluation	57
9.1	Circuit Verification	57
9.2	Proof Generation and Checking	62
10	Conclusion	65
II	Papers	67
A	Incremental Column-Wise Verification of Arithmetic Circuits Using Computer Algebra	71
A.1	Introduction	72
A.2	Algebra	74
A.3	Ideals associated to Circuits	78
A.4	Optimizations	82
A.5	Variable Elimination	85
A.6	Order	91
A.7	Incremental Column-Wise Checking	92
A.8	Incremental Equivalence Checking	94
A.9	Engineering	97
A.10	Experiments	100
A.11	Conclusion	106
A.12	Acknowledgements	106
B	A Practical Polynomial Calculus for Arithmetic Circuit Verification	109
B.1	Introduction	109
B.2	Preliminaries	111
B.3	Practical Algebraic Calculus	114
B.4	Circuit verification using Computer Algebra	115
B.5	Engineering	117
B.6	Experiments	120
B.7	Conclusion	124
C	Verifying Large Multipliers by Combining SAT and Computer Algebra	127
C.1	Introduction	127
C.2	Specifying Multiplier Circuits	128
C.3	Algebra	130
C.4	D-Gröbner bases	134

C.5	Variable elimination	136
C.6	Combining SAT and Computer Algebra	137
C.7	Experiments	141
C.8	Conclusion	144
D	SAT, Computer Algebra, Multipliers	147
D.1	Introduction	147
D.2	Algebraic approach	148
D.3	SAT	152
D.4	AMulet	154
D.5	Proof Generation	159
D.6	Proof Size	161
D.7	Conclusion	167
E	From DRUP to PAC and Back	169
E.1	Introduction	170
E.2	Preliminaries	170
E.3	From DRUP to PAC	173
E.4	From PAC to DRUP	175
E.5	Experiments	178
E.6	Conclusion	178
F	The Proof Checkers Pacheck and Pastèque for the Practical Algebraic Calculus	181
F.1	Introduction	181
F.2	Practical Algebraic Calculus	182
F.3	Pacheck	184
F.4	Pastèque	185
F.5	Evaluation	186
F.6	Conclusion and Future Work	188
F.7	Appendix	189
	Bibliography	191

Part I

Prologue

Chapter 1

Introduction

“I would hope that computers and computer analysis would lose some of the aura of invincibility with which they have been treated. Computer generated results need to be treated with some enlightened skepticism. No system or microprocessor can be expected to produce results which are absolutely reliable. ”

–Thomas R. Nicely

Digital circuits carry out logical operations, which makes them an important element 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, which represent simple Boolean functions, such as negation (NOT), conjunction (AND), or disjunction (OR). These logic gates can be combined to build more complex logical operations. A subclass of digital circuits are combinational logic circuits where the output is a function of the present 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*.

Formal verification is used to prove or disprove the correctness of a given software or hardware 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 properties. The different formal verification approaches are distinguished by the mathematical formalisms used in the verification process.

Formal verification of arithmetic circuits is important to help to prevent issues like the famous Pentium FDIV bug that was detected by Thomas R. Nicely 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, the problem of formally verifying arithmetic circuits, and especially multiplier circuits, is still considered to be hard.

Up to now several solving techniques have been developed for multiplier verification. 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 [12]. However, the results indicated that verifying miters of multipliers and other ring properties after encoding them into CNF needs exponential sized resolution proofs [14], which implies exponential run-time of CDCL SAT solvers. This conjecture is neglected in theory in [7], where it was shown that ring properties do admit polynomial sized resolution proofs. However, this theoretical result still needs to be transferred into practice.

Another approach is based on the usage of theorem provers, such as ACL2 [68]. Theorem provers in combination with SAT are able to certify industrial multipliers [54], however, this technique is not fully automated and requires a lot of domain knowledge, since the underlying proof system is based on a problem-specific set of axioms and inference rules. Methods based on term rewriting [97] require domain knowledge too and thus are not fully automated either.

Approaches based on bit-level reverse engineering [86, 95] use arithmetic bit-level representations, which are extracted from the given gate-level netlists. This technique is able to verify simple multipliers, but fails to verify non-trivial multiplier architectures.

The first technique that was shown to detect the Pentium bug is based on binary decision diagrams [22], more precisely on binary moment diagrams (BMDs) [29] and variants [30], since their size remains linear in the number of input bits of a multiplier. However, this approach requires structural knowledge of the multipliers [24, 29]. It is important to determine the order in which BMDs are built, because it has tremendous influence on the size and thus performance.

The currently most effective technique for fully automated verification of multipliers is based on computer algebra, e.g., [31, 62, 80]. In this method all logic gates of the circuit and the specification are represented by polynomials. If the gate polynomials are ordered according to their reverse topological appearance in the circuit, they automatically form a Gröbner basis [25]. As a consequence, the question whether a circuit implements a correct multiplier can be answered by reducing the specification polynomial by the Gröbner basis. The multiplier is correct if and only if the reduction returns zero. The main issue of the algebraic approach is that without preprocessing the size of intermediate reduction results increases drastically.

The aim of this thesis is to investigate and improve formal verification of multiplier circuits using computer algebra to make it practically applicable for non-trivial and optimized multiplier designs. We propose a simple and precise mathematical formalization of the algebraic approach for arithmetic circuit verification, including rigorous proofs of soundness and completeness. We first assume that the coefficient domain is a field in order to use the Gröbner basis theory over fields [25]. In the following, we generalize the approach of circuit verification to be applicable in more general polynomial rings that allow modular reasoning. Modular reasoning allows us to verify not only signed and

unsigned integer multipliers, but also truncated multipliers, where the most significant output bits are discarded. These multipliers are used for example in SMT-LIB [5].

Furthermore, we investigate possible reasons of the monomial blow-up in intermediate reduction results. Based on these results we develop reasoning techniques that overcome this issue. One of our technical contributions is a new incremental column-based verification approach for multipliers. In this method the multiplier circuit is divided into several slices and 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.

We further develop techniques based on variable elimination, which rewrite and simplify the Gröbner basis. In the first version of our rewriting techniques we extract certain subcircuits from the circuit and eliminate the internal variables of these subcircuits from the Gröbner basis. For this optimization we introduce the necessary theory and present a technical theorem that allows us to rewrite only local parts of the Gröbner basis in such a way that the result is again a Gröbner basis. Based on these results we are able to formalize a more general version of variable elimination, which does not require to identify syntactic patterns in the circuits.

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 explosion of the intermediate reduction results. On the other hand, equivalence checking of adders is easy for SAT. Based on this observation we combine SAT and computer algebra in our verification technique. We detect whether a multiplier contains a complex final stage adder. If necessary, we replace the complex adder by a simple ripple-carry adder. The correctness of the replacement step is verified by SAT solvers and the rewritten multiplier is verified by computer algebra techniques.

Initially, we use existing computer algebra systems (CAS) to apply Gröbner basis reduction, but these systems are designed for general purposes and thus are slow for our application. Hence, we implement a dedicated reduction engine AMULET, which is tailored to the specific structure of the problem. Implementing our own tool gives a speed-up of three orders of magnitude compared to CAS, cf. Chapter 9.

After applying all these sophisticated techniques, we are able to verify the correctness of multipliers. However, the result of automated reasoning tools might not be error-free. A common approach to increase confidence in the verification results consists of generating proofs, which are checked by independent proof checkers. In order to validate verification techniques based on computer algebra, we show how the abstract polynomial calculus [34] can be instantiated to yield a practical algebraic calculus (PAC), which can be checked efficiently. Proofs in this format can be obtained as by-products in AMULET. We implement independent proof checking tools PACTRIM and PACHECK, where PACHECK is an extension of PACTRIM and supports an extended version of PAC.

Combining SAT and computer algebra has the effect that two proof certificates in different proof formats (DRUP [47] and PAC) are generated. We also investigate how these proof systems can be merged in order to derive one single proof certificate.

1.1 Outline

This thesis is structured as a cumulative dissertation and consists of two parts, which allows us to present our work as it was accepted at international conferences and workshops without changing the exposition. The new content of this thesis, contained in Part I, is clearly separated from already published work, which is contained in Part II.

In Part I of the thesis we discuss the contributions of the published Papers A–F, which are included in Part II of this thesis. At first we introduce the background and related work in Chap. 2. In Chap. 3–8 we revisit and reflect on the ideas of Papers A–F. We put our work into context and summarize and analyze the contributions of each paper. We highlight the advantages, but also reflect on downsides, which we noticed in retrospective and lead to new research ideas in later work. In Chap. 9 we give a comprehensive evaluation of our developed techniques that shows the improvement over time. Furthermore, we compare our tools to the current state of the art of related work.

Part II of the thesis, i.e., Paper A–F, consists of six papers [61, 62, 63, 64, 66, 90], where the author of this thesis is the main author¹. Papers [61, 62, 63, 90] are peer-reviewed, [64] is an invited paper published in post-proceedings and [66] is currently under review. The status of the papers is clearly indicated at the beginning of each chapter in Part II. The included papers contain small modifications compared to the original publications such as fixed typos and layout changes to evolve a consistent layout and bibliography. Fixes are clearly stated at the beginning of each chapter. The content of the papers is not modified.

1.2 Contributions

The author of this thesis is the main author of Papers A–F. However, none of the work would have been possible without the help of others. In the following paragraphs we clearly point out the contributions of the author of this thesis.

Paper A. [61] *Incremental Column-Wise Verification of Arithmetic Circuits Using Computer Algebra* with Armin Biere and Manuel Kauers. To be published in the Special Issue on Formal Methods in Computer-Aided Design of the International Journal on Formal Methods in System Design (FMSD) and is currently available as “Online First” article. This paper summarizes and extends work presented in [17, 89, 91].

In Paper A, we give a rigorous formalization of the algebraic verification approach. We include a new incremental column-wise verification approach and further improve this algorithm by adding rewriting techniques based on variable elimination.

The formalization was the contribution of all authors and was described by D. Kaufmann. The incremental algorithm was a result of discussions between A. Biere and D. Kaufmann. The rewriting techniques were developed by D. Kaufmann and described with contributions from M. Kauers. The implementation in Mathematica [102] and

¹Please note, some of the work is published under the author’s maiden name “Ritirc”.

Singular [38] was developed by D. Kaufmann. The first version of the tool AIG-MULTOPOLY, as it is published in [89] was implemented by A. Biere and is since then [17, 61, 91] extended and maintained by D. Kaufmann. The experimental analysis was performed by D. Kaufmann. The co-authors further contributed with discussions and proofreading Paper A.

Paper B. [90] *A Practical Polynomial Calculus for Arithmetic Circuit Verification* with Armin Biere and Manuel Kauers. In Proceedings of the 3rd Workshop on Satisfiability Checking and Symbolic Computation (SC’2) co-located with Federated Logic Conference (FLOC 2018), pages 61–76, Oxford, United Kingdom, 2018.

Paper B presents an algebraic proof calculus (PAC), based on the polynomial calculus, which allows derivation of proof certificates. Proofs in this format can be checked by a CAS or our independent proof checker PACTRIM.

The formalization of the calculus was the result of discussions of all authors and was described by D. Kaufmann. Generating and checking PAC proofs using a CAS was implemented by D. Kaufmann. The first version of PACTRIM, as it is published in Paper B was implemented by A. Biere and is since then extended and maintained by D. Kaufmann. The experimental analysis was performed by D. Kaufmann. The co-authors further contributed with discussions and proofreading Paper B.

Paper C. [62] *Verifying Large Multipliers by Combining SAT and Computer Algebra* with Armin Biere and Manuel Kauers. In Proceedings of the 19th International Conference on Formal Methods in Computer Aided Design (FMCAD 2019), pages 28–36, San Jose, CA, USA, 2019.

In Paper C we generalize the algebraic verification approach to be applicable in more general polynomial rings. We combine SAT and computer algebra to substantially improve automated reasoning for circuit verification. Furthermore, we present a rewriting technique that does not involve syntactic pattern matching and implement our dedicated reduction engine AMULET.

The method for combining SAT and computer algebra was developed by D. Kaufmann. The incentive to model circuits in more general polynomial ring was given by D. Kaufmann and was described by all authors. The rewriting technique was established by D. Kaufmann. AMULET was implemented and is maintained by D. Kaufmann. The experimental analysis was performed by D. Kaufmann. The co-authors further contributed with discussions and proofreading Paper C.

Paper D. [64] *SAT, Computer Algebra, Multipliers* with Armin Biere and Manuel Kauers. Invited paper in the Post-Proceedings of the 5th and 6th Vampire Workshops, Vampire 2018 and Vampire 2019, pages 1–18, Lisbon, Portugal, 2019.

Paper D extends Paper C and gives a rigorous system description of our tool AMULET. We discuss the algorithms and present how proof certificates are generated in AMULET. Furthermore, we examine the proof size of certain multiplier benchmarks.

The system description and the discussion on proof size were performed by D. Kaufmann. The co-authors further contributed with discussions and proofreading Paper D.

Paper E. [63] *From DRUP to PAC and Back* with Armin Biere and Manuel Kauers. To be published in Proceedings of the Design, Automation & Test in Europe Conference (DATE 2020), 4 pages, Grenoble, France, 2020.

In the work of Paper C, two proof certificates in different proof formats are generated. For SAT a DRUP proof is generated and for the algebraic approach a PAC proof is generated. In Paper E we investigate how the proof formats can be merged in order to generate only one single proof certificate.

The incentive for translating PAC proofs into DRUP proofs was given by A. Biere. The translation of DRUP proofs into PAC proofs was developed by D. Kaufmann. The techniques in Paper E were described by D. Kaufmann. All developed tools in Paper E were implemented by D. Kaufmann. The experimental analysis was performed by D. Kaufmann. The co-authors further contributed with discussions and proofreading Paper E.

Paper F. [66] *The Proof Checkers Pacheck and Pastèque for the Practical Algebraic Calculus* with Mathias Fleury and Armin Biere. Submitted.

In Paper F we present our proof checkers PACHECK and PASTÈQUE. The checker PACHECK checks algebraic proofs more efficiently than PASTÈQUE, which is formally verified using the proof assistant Isabelle/HOL. Furthermore, we extend the practical algebraic calculus of Paper B by adding deletion and extension rules and introduce a more compact syntax.

The incentive for extending the practical algebraic calculus was given by D. Kaufmann. The calculus was described by D. Kaufmann. PACHECK was implemented by D. Kaufmann. PASTÈQUE was implemented and verified in Isabelle/HOL by M. Fleury. The experimental evaluation was performed by D. Kaufmann. The co-authors further contributed with discussions and proofreading Paper F.

Chapter 2

Background

This section provides an overview of the background of this thesis. We present the basic structure of multiplier circuits (Sect. 2.1) and give an introduction of computer algebra (Sect. 2.2) and arithmetic circuit verification using computer algebra (Sect. 2.3). Furthermore, we outline SAT (Sect. 2.4) and algebraic proof systems (Sect. 2.5) and conclude this section by discussing related work (Sect. 2.6).

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 usually realized by logic gates, such as NOT, AND, and OR. 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 thesis, we consider gate-level integer multipliers with input bits a_0, \dots, a_{n-1} , $b_0, \dots, b_{n-1} \in \{0, 1\}$ and $2n$ output bits $s_0, \dots, s_{2n-1} \in \{0, 1\}$. If the circuit represents multiplication of unsigned integers, the multiplier is correct if and only if for all possible inputs the following specification holds:

$$\mathcal{L} = - \sum_{i=0}^{2n-1} 2^i s_i + \left(\sum_{i=0}^{n-1} 2^i a_i \right) \left(\sum_{i=0}^{n-1} 2^i b_i \right) = 0 \quad (2.1)$$

Example 2.1. Figure 2.1a 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 $-8s_3 - 4s_2 - 2s_1 - s_0 + (2b_1 + b_0)(2a_1 + a_0) = 0$.

A common representation of combinational circuits is the encoding as an And-Inverter-Graph (AIG) [70]. An AIG is a special case of a directed acyclic graph, and consists only of two-input nodes representing logical conjunction. The edges of an AIG may contain a marking that indicates logical negation. The AIG representation of a circuit is usually larger, i.e., contains more nodes, than the gate-level representation, but is very efficient to manipulate. In this thesis all circuits are given as AIGs. Figure 2.1b shows the AIG representation of the multiplier depicted in Fig. 2.1a.

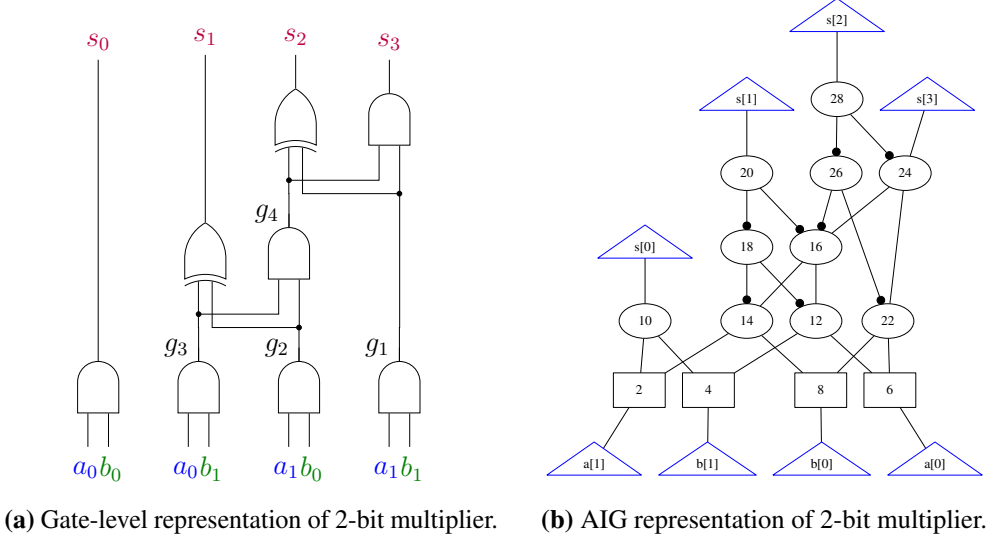


Figure 2.1: Graph representation of 2-bit multipliers.

The space and time complexity of a multiplier circuit depends highly on its architecture. Multiplier circuits can generally be decomposed into three components [85]. 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 [85].

In the second component, *partial product accumulation* (PPA), multi-operand addition is performed using full- and half-adders to reduce the partial products to two layers. Well-known accumulation structures are array accumulation, diagonal accumulation, Wallace trees, or compressor trees [85].

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 adders 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 [85].

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

Example 2.2. Figure 2.2 shows two simple multiplier architectures with input bit-width 4. In both circuits the PPG uses AND-gates, i.e., $p_{ij} = a_i \wedge b_j$. In “btor”-multipliers [83], which are shown on the left side, the partial products are accumulated using an array structure. The “sp-ar-rc”-multipliers [53] (also called “sparrc” in this thesis), which are depicted on the right side, use a diagonal structure. In both multipliers, the FSA is a ripple-carry adder, which is highlighted in red.

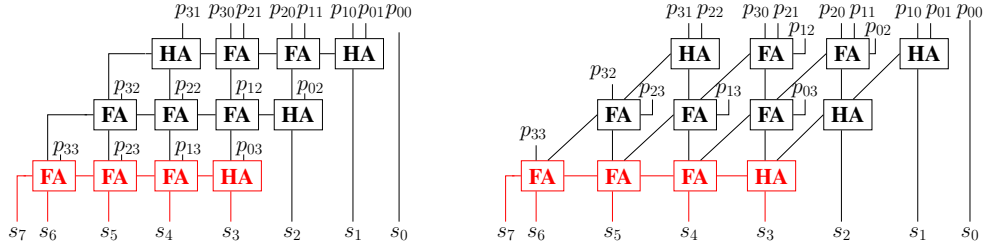


Figure 2.2: Architecture of “btor” (left) and “sp-ar-rc” (right) multipliers.

2.2 Algebra

In this section we introduce algebraic concepts, following [35]. Throughout this section let $\mathbb{K}[X] = \mathbb{K}[x_1, \dots, x_n]$ denote the ring of polynomials in variables x_1, \dots, x_n with coefficients in a field \mathbb{K} .

- A *term* τ is a product of the form $\tau = x_1^{e_1} \cdots x_n^{e_n}$ for certain $e_1, \dots, e_n \in \mathbb{N}$. 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 $\sigma_1 < \sigma_2$ iff 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 (w.r.t. the chosen order \leq) is called the *leading term* and denoted by $\text{lt}(p)$.
- If $p = \alpha\tau + \cdots$ and $\text{lt}(p) = \tau$, then $\text{lc}(p) = \alpha$ is called the *leading coefficient* and $\text{lm}(p) = \alpha\tau$ is called the *leading monomial* of p . We call $p - \alpha\tau$ the *tail* of p .
- For a given set of polynomials $P = \{p_1, \dots, p_m\} \subseteq \mathbb{K}[X]$, a *model* is a point $u = (u_1, \dots, u_n) \in \mathbb{K}^n$ such that for all $p_i \in P$ we conclude that $p_i(u_1, \dots, u_n) = 0$.
- A nonempty subset $I \subseteq \mathbb{K}[X]$ 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 $P = \{p_1, \dots, p_m\} \subseteq \mathbb{K}[X]$ is called a *basis* of I if $I = \{q_1p_1 + \cdots + q_m p_m \mid q_1, \dots, q_m \in \mathbb{K}[X]\}$. We say I is *generated* by P and write $I = \langle P \rangle$.
- A basis $P = \{p_1, \dots, p_m\}$ of an ideal $I \subseteq \mathbb{K}[X]$ is called a *Gröbner basis* (w.r.t. the fixed order \leq) iff $\forall q \in I \exists p_i \in P : \text{lm}(p_i) \mid \text{lm}(q)$.
- Every ideal of $\mathbb{K}[X]$ has a Gröbner basis, and there is an algorithm which, given an arbitrary basis of an ideal, computes a Gröbner basis of it [25].

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 $P = \{p_1, \dots, p_m\} \subseteq \mathbb{K}[X]$, decide whether q belongs to the ideal generated by p_1, \dots, p_m . If $\{p_1, \dots, p_m\}$ is a Gröbner basis, then the question can be answered using a multivariate version of polynomial division with remainder. The polynomial q belongs to $\langle P \rangle$ if and only if the remainder of division of q by P is zero. More facts of Gröbner bases are:

- Let $q \in \mathbb{K}[X]$ and $P = \{p_1, \dots, p_m\} \subseteq \mathbb{K}[X]$. The remainder r of the division of q by P is a polynomial such that $q - r$ is contained in the ideal generated by P and r is *reduced* w.r.t. P , which means it does not contain any term that is a multiple of one of the leading terms $\text{lt}(p_1), \dots, \text{lt}(p_m)$.
- Let $P \subseteq \mathbb{K}[X] \setminus \{0\}$, and define

$$\text{spol}(p, q) := \text{lcm}(\text{lt}(p), \text{lt}(q)) \left(\frac{p}{\text{lm}(p)} - \frac{q}{\text{lm}(q)} \right)$$

for all $p, q \in \mathbb{K}[X] \setminus \{0\}$, with lcm the least common multiple. Then P is a Gröbner basis if and only if the remainder of the division of $\text{spol}(p, q)$ by P is zero for all pairs $(p, q) \in P \times P$.

- If $p, q \in \mathbb{K}[X] \setminus \{0\}$ are such that their leading terms $\text{lt}(p), \text{lt}(q)$ have no variables in common, then the division of $\text{spol}(p, q)$ with $\{p, q\}$ has remainder zero. This property is known as *Buchberger's product criterion*.

In this section we restricted \mathbb{K} to be a field. We present the algebraic background for coefficient domains R , where R is a commutative ring with unity, in Paper C, where we generalize the theory to be applicable in more general polynomial rings.

2.3 Circuit Verification using Computer Algebra

In this section we give a short introduction of circuit verification using computer algebra, a rigorous formalization of this approach can be found in Paper A. We consider circuits with $2n$ inputs a_0, \dots, a_{n-1} and b_0, \dots, b_{n-1} , $2n$ 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 semantic of each circuit gate implies a polynomial relation among the input and output variables, such as the following ones:

$$\begin{array}{lll} u = \neg v & \text{implies} & 0 = -u + 1 - v \\ u = v \wedge w & \text{implies} & 0 = -u + vw \\ u = v \vee w & \text{implies} & 0 = -u + v + w - vw \\ u = v \oplus w & \text{implies} & 0 = -u + v + w - 2vw. \end{array} \quad (2.2)$$

We call these polynomials *gate polynomials* or *gate constraints*. Let $G \subseteq R$ be the set of polynomials, which contains for each gate of the given circuit the corresponding polynomial of Eqn. (2.2).

To enforce that our variables are Boolean and can have only the values 0 and 1, we add for each variable $x \in X$ the relation $x(x - 1) = 0$. In Papers A and B we call these polynomials *field polynomials*. We renamed the definition in Paper C to *Boolean value constraints*, because the term “field” may suggest a connection to the coefficient field \mathbb{K} .

We order the set of terms according to a lexicographic term order, where the output variable of a gate is always greater than the input variables of a gate. Such an ordering is called *reverse topological term order* [78]. Because of Buchberger’s product criterion and the structure of the gate polynomials, the gate polynomials together with the Boolean value constraints define a Gröbner basis for the ideal generated by the gate polynomials and Boolean value constraints. Thus the correctness of the circuit can be shown by reducing the specification \mathcal{L} by the gate polynomials using polynomial reduction and checking whether the result is zero.

2.4 SAT

We briefly introduce the SAT problem, following [47].

- A *literal* l is either a positive Boolean variable x or its negation \bar{x} .
- A *clause* C is a finite disjunction of literals. If a clause contains only one literal, we call it a *unit clause*.
- 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 $\tau(x) = v \Leftrightarrow \tau(\bar{x}) = \neg v$, where $\neg \mathbf{t} = \mathbf{f}$ and $\neg \mathbf{f} = \mathbf{t}$.

A formula evaluates to \mathbf{t} if and only if every clause in the formula evaluates to \mathbf{t} . A clause C evaluates to \mathbf{t} if there exists a literal $l \in C$ 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 \mathbf{t} . If such an assignment exists, the formula is *satisfiable*, otherwise it is *unsatisfiable*.

A clause C is *redundant* w.r.t. a formula F , if $F \wedge C$ is satisfiable iff F is satisfiable. Redundant clauses are for example derived using *resolution* [92]: Given two clauses $C_1 = (a \vee x_0 \vee \dots \vee x_m)$ and $C_2 = (\bar{a} \vee y_0 \vee \dots \vee y_n)$ the clause $C = (x_0 \vee \dots \vee x_m \vee y_0 \vee \dots \vee y_n)$ can be resolved.

A common technique used in SAT solvers is called *unit propagation*: If a formula F contains a unit clause $C = l$, remove all clauses containing l and all occurrences of \bar{l} .

If a formula is satisfiable a satisfying assignment is a witness. However, if the formula is unsatisfiable more involved reasoning is required to derive proofs of unsatisfiability, also called *refutation*. This is done by showing that the empty clause is redundant. Providing certificates of unsatisfiability is mandatory in the SAT Competitions since 2013.

Standard refutation proof formats are either resolution proofs or clausal proofs. Clausal proofs are easier to generate and are more compact than resolution proofs. The most basic clausal proof format is *reverse unit propagation* (RUP) [41]. Let \bar{C}

denote the negation of a clause C . If for example $C = a \vee b \vee \bar{x}$ then $\bar{C} = \bar{a} \wedge \bar{b} \wedge x$. We say C is a *RUP clause* if $F \wedge \bar{C}$ evaluates to **f** only by *unit propagation*. A RUP proof is a sequence of RUP clauses containing the empty clause.

A *delete reverse unit propagation* (DRUP) [49] proof extends RUP by adding deletion information to decrease the cost of proof validation [99]. Clausal DRUP proofs are checked through unit propagation. As a side effect a resolution proof can be produced.

2.5 Algebraic Proof Systems

Algebraic proof systems reason over polynomials in $\mathbb{K}[X]$, where \mathbb{K} is a field and the variables $X = \{x_1, \dots, x_l\}$ represent Boolean values. Thus it holds for each $x_i \in X$, that $x_i^2 - x_i = 0$. The aim of an algebraic proof is to derive a refutation, i.e., derive that a given set of polynomials $P = \{p_1, \dots, p_m\} \subseteq \mathbb{K}[X]$ together with the Boolean value constraints $B(X) = \{x_i^2 - x_i \mid x_i \in X\}$ has no common roots. In algebraic terms this means to show that the constant polynomial $1 \in \langle P \cup B(X) \rangle$.

In the following we present two very common proof formats, which are able to derive a refutation using algebraic reasoning methods.

2.5.1 Polynomial Calculus

The first proof system is the *polynomial calculus* (PC) [34]. A proof in PC is a sequence of proof rules $R = (r_1, \dots, r_n)$, which model the properties of an ideal. A PC proof is a correct refutation when the constant polynomial 1 is derived. Each rule has the following form:

Axiom	$\frac{}{p_i} \quad p_i \in P$
Boolean Axiom	$\frac{}{x_j^2 - x_j} \quad x_j \in X$
Linear combination	$\frac{p \quad q}{\alpha p + \beta q} \quad \begin{array}{l} p, q \text{ appearing earlier in the proof} \\ \alpha, \beta \in \mathbb{K} \end{array}$
Multiplication	$\frac{p}{\tau p} \quad \begin{array}{l} p \text{ appearing earlier in the proof} \\ \tau \text{ any term} \end{array}$

The *polynomial calculus with resolution* (PCR) [3] extends PC by adding a negation rule, which models that a new variable \bar{x}_j is the logical negation of x_j .

$$\text{Negation} \quad \frac{}{x_j + \bar{x}_j - 1}$$

Since \bar{x}_j is a newly added variable, PCR operates in the ring $\mathbb{K}[x_1, \dots, x_l, \bar{x}_1, \dots, \bar{x}_l]$.

Adding new variables makes it possible to produce shorter proofs, as we will discuss later in Sect. 7.1.2.

2.5.2 Nullstellensatz

The second algebraic proof system we consider is Nullstellensatz [6]. A Nullstellensatz refutation of a set of polynomials $P = \{p_1, \dots, p_m\} \subseteq \mathbb{K}[X]$ is an equality

$$\sum_{i=1}^m q_i p_i + \sum_{j=1}^l r_j (x_j^2 - x_j) = 1, \quad (2.3)$$

with $q_i, r_j \in \mathbb{K}[X]$.

2.6 Related Work

In recent years great progress has been achieved on verifying (integer) multiplier circuits using computer algebra. We consider the work of Sayed-Ahmed et al. [93, 94], Mahzoon et al. [79, 80] and, Yu et al. [31, 32, 103, 104, 106] the most related work, as their research also focuses on verification of integer multipliers using computer algebra. The work of Lv et al. [77, 78] focuses on Galois field multipliers. In contrast to word-level verification the work of Brickenstein et al. [20] proposes bit-level equivalence checking of multipliers.

Sayed-Ahmed et al. [93, 94] The authors of [93] designed a dedicated polynomial reduction engine and also present various syntactic rewriting optimizations, which makes their algebraic technique scale to large non-trivial multiplier designs of various architectures. In follow-up work [94] they propose an algebraic variant of combinational equivalence checking, also based on Gröbner basis theory. It is similar to SAT sweeping [70], and compares circuits bit-wise, i.e., output by output. However, their tools of [93, 94] are not available, nor are details about the experiments.

Mahzoon et al. [79, 80] Vanishing monomials are monomials that occur in intermediate results and evaluate to zero during the reduction process. In [79] Mahzoon et al. discuss why vanishing monomials occur and propose a method, which searches for converging gate cones and allows local cancellation of the vanishing monomials before global backward rewriting is applied in order to prevent an explosion of the intermediate reduction results. In [80] the authors present a further optimization, which is able to detect so-called “atomic blocks” in order to speed-up rewriting by reducing the search space for finding converging gates. This approach is very successful in verifying a large variety of multiplier architectures but is an order of magnitude slower than our approach of Paper C, as we will show in Chap. 9.

Yu et al. [31, 32, 103, 104, 106] In [32, 103] the authors use *function extraction*, a similar algebraic approach to Gröbner basis reduction. The word-level output of the circuit is rewritten using the gate relations and the goal is to derive a unique polynomial representation of the circuit inputs. In order to verify correctness of the circuit, this polynomial is then compared to the circuit specification. In follow-up work [31, 106]

full- and half-adders are identified and replaced by polynomials to simplify the set of polynomials. Their technique is able to handle very large clean multipliers efficiently but fails on slightly optimized multiplier architectures. The authors also extended their work to Galois field multipliers [104].

Brickenstein et al. [20] The authors of [20] design a framework for Gröbner basis computations with Boolean polynomials. They introduce a specialized data structure based on zero-suppressed binary decision diagrams, which allows efficient manipulation of Boolean polynomials. Their approach is able to apply bit-wise equivalence checking of integer multiplier circuits, but handles only small-size multipliers.

Lv et al. [77, 78] The authors of [77, 78] use an algebraic approach to verify Galois field multipliers. The multiplier circuit is modeled as a polynomial system in $\mathbb{F}_{2^k}[X]$ and consequently Gröbner basis theory over Galois fields is used to derive the correctness of the multiplier.

Chapter 3

Paper A: Incremental Column-Wise Verification of Arithmetic Circuits Using Computer Algebra

In Paper A we first give a comprehensive formalization of arithmetic circuit verification using computer algebra and prove soundness and completeness. The general idea of arithmetic circuit verification using computer algebra is to model each gate by corresponding polynomial equations, called *gate constraints*. Correctness of the circuit is derived by showing that the specification \mathcal{L} , like Eqn. (2.1), represented as a polynomial, is contained in the ideal generated by the gate constraints. In this section we consider circuits with $2n$ inputs $a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1}$ and $2n$ outputs s_0, \dots, s_{2n-1} .

In order to optimize the practical application of arithmetic circuit verification using computer algebra, we present a novel incremental column-wise verification algorithm, which allows us to split the verification problem into smaller, more manageable sub-problems. The multiplier is partitioned into slices, such that each slice contains exactly one output bit. For each slice we derive corresponding slice-wise specifications, which in contrast to the word-level specification, only contain a linear number of partial products. Correctness of the multiplier is derived by verifying each slice incrementally.

To further enhance the practical approach we present a preprocessing technique, which rewrites the Gröbner basis implied by the gate constraints. We search and identify certain structures, namely full- and half-adders, XOR-gates and gates with only one parent, in the input AIG. Internal variables of these patterns are eliminated from the Gröbner basis. We prove a technical theorem that allows us to rewrite only local parts of the Gröbner basis without violating the Gröbner basis property. By eliminating internal nodes of full- and half-adders and XOR-gates, we only represent the specification of these structures in the Gröbner basis instead of the direct translation of all internal nodes. For full- and half-adders these polynomials are linear and thus significantly reduce the blow-up of the intermediate reduction results.

We implement a tool, called AIGMULTOPOLY, which translates multipliers given as AIGs into a set of polynomials and applies the described preprocessing steps. The output file is passed to the CAS Mathematica [102] or Singular [38] that apply the polynomial reduction. Our experiments show that by using these techniques we are able to verify simple multipliers but fail to verify complex multipliers.

We furthermore present how the incremental verification algorithm can be generalized to allow equivalence checking of circuits. In equivalence checking a circuit is compared to a so-called golden model, which is a circuit that is known to be correct. The benefit of equivalence checking, in comparison to circuit verification is that the specification of the circuit does not need to be known. In our approach we apply equivalence checking as follows. Let $S = \sum_{i=0}^n 2^i s_i$ be the output bit-vector of the circuit whose correctness is in question and let $S' = \sum_{i=0}^n 2^i s'_i$ be the output of the golden model. The gates of both circuits are encoded as polynomials and we check whether the polynomial $S - S' = 0$ is contained in the ideal generated by the circuit constraints and the Boolean value constraints. However, this has the effect that actually both circuits are verified simultaneously. It can be seen in the experiments of Paper A that equivalence checking of multipliers needs twice as much time than verifying one single multiplier. Since we are able to define a specification for multipliers, we favor direct verification of multiplier circuits over equivalence checking with the golden model and we do not apply equivalence checking of circuits in follow-up work of circuit verification.

3.1 Polynomial Ring

Initially we wanted to work in the ring $\mathbb{Z}[X]$, because \mathbb{Z} is the ring whose multiplication we want to describe. However, \mathbb{Z} is not a field and we cannot use the basic Gröbner bases theory over fields [25]. Since \mathbb{Q} is a field, which contains \mathbb{Z} , we fixed the polynomial ring to $\mathbb{Q}[X]$ in Paper A. Additionally, the reduction algorithms in Mathematica and Singular are much slower for $\mathbb{Z}[X]$ than for $\mathbb{Q}[X]$. Furthermore, we noticed that because the leading coefficients of the polynomials are all -1 , our whole computation stays in the ring $\mathbb{Z}[X]$ anyhow.

In retrospective, modeling the verification problem in $\mathbb{Q}[X]$ is not optimal and limits the power of our procedure. The reason is that certain multiplier architectures invoke chains of XOR-gates to compute the most significant output bit s_{2n-1} . For example, this always happens in our benchmarks, when Booth encoding is used to generate the partial products. The corresponding coefficient of s_{2n-1} in the word-level specification is 2^{2n-1} . Hence, reduction by a polynomial, which encodes an XOR-gate generates a monomial $-2^{2n}vw$, cf. Eqn. (2.2), where either v or w represents again the output variable of an XOR-gate. Thus reducing s_{2n-1} by the polynomials encoding the XOR-gates yields multiple monomials with coefficients that are multiples of 2^{2n} , which will reduce to zero later in the verification procedure. However, these non-linear monomials have to be rewritten first and thus increase the results of the intermediate reduction results until they are canceled.

We discuss in Paper C that it is a better choice to select the ring $\mathbb{Z}_{2^l}[X]$, where l is the number of output bits. This has the effect that any monomial, which is a multiple of 2^l , is directly eliminated. We prove that we maintain soundness and completeness by choosing l as the number of output bits. Selecting $\mathbb{Z}_{2^l}[X]$ does not only make verification of unsigned and signed integer multipliers much more efficient, but also allows verification of truncated multipliers, where the n most significant bits are discarded.

polynomial ring	sec	max	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0
$\mathbb{Q}[X]$	0.07	2 264	2	25	57	94	358	157	36	36
$\mathbb{Z}_{2^{2n}}[X]$	0.00	36	2	7	12	14	15	14	6	5

Table 3.1: Number of monomials in intermediate results for 4-bit “bp-ar-rc” multiplier.

We show in Paper C that the ideal membership problem in $\mathbb{Z}_{2^l}[X]$ for an ideal I can be converted into an ideal membership problem in the ring $\mathbb{Z}[X]$. Whenever we want to decide whether a polynomial $q \in I \subseteq \mathbb{Z}_{2^l}[X]$ we can instead check whether $q \in I + \langle 2^l \rangle \subseteq \mathbb{Z}[X]$. For the latter we can use the theory of D-Gröbner bases [9, 76].

Example 3.1. Consider a 4-bit “bp-ar-rc”-multiplier [53]. This multiplier is very similar to the “sp-ar-rc”-multiplier shown in Fig. 2.2, except that the partial-products are generated using a Booth encoding instead of simple AND-gates. We apply our incremental column-wise verification algorithm in $\mathbb{Q}[X]$ and $\mathbb{Z}_{2^{2n}}[X]$ using our tool AMULET introduced in Paper C. Table 3.1 shows the total verification time and the maximum size, i.e., number of monomials, of the intermediate reduction results. Furthermore, we list the size of the incremental slice-wise specifications.

Circuits operate over binary values. Thus, it would theoretically be possible to model the verification problem in the ring $\mathbb{Z}_2[X]$. The gate constraints in the ring \mathbb{Z}_2 have the following form, which gives a linear representation for XOR-gates.

$$\begin{aligned}
u = \neg v & \quad \text{implies} \quad 0 = u + 1 + v \\
u = v \wedge w & \quad \text{implies} \quad 0 = u + vw \\
u = v \vee w & \quad \text{implies} \quad 0 = u + v + w + vw \\
u = v \oplus w & \quad \text{implies} \quad 0 = u + v + w
\end{aligned} \tag{3.1}$$

Furthermore, there is no need to add the Boolean constraint equation because it automatically follows in $\mathbb{Z}_2[X]$ that $x_i^2 = x_i$ for $x_i \in X$.

While modeling the gate constraints is straightforward, the problem is to model the word-level specification, since the specification we want to describe contains coefficients that are multiples of 2. Recall the specification of Eqn. 2.1 we consider is

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

Defining the specification in $\mathbb{Z}_2[X]$ actually reduces reasoning from word-level to reasoning on bit-level. A possibility would be to define a bit-level specification for each output bit separately, that is, define for each output bit s_i a function f_i over the input bits, such that $-s_i + f_i(a_{n-1}, \dots, a_0, b_{n-1}, \dots, b_0) = 0$. We are able to experimentally generate the bit-level specifications for 4-bit multipliers, but our approach already times out for input bit-width 5.

Example 3.2. Consider “btor”-multipliers, cf. Fig. 2.2, of input bit-width 3 and 4. Table 3.2 shows the number of monomials in the corresponding bit-level specifications.

mult	n	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
btor	3	2	3	5	9	10	4		
btor	4	2	3	5	11	25	38	42	20

Table 3.2: Length of bit-level specifications.

In favor of our observation, the works of [23, 33, 101] show that binary decision diagrams representing the middle bit of multipliers have exponential size. On the contrary, the authors of [19] are able to apply equivalence checking of multipliers on bit-level for multipliers up to 16 bit. They have the conjecture that their experiments do not admit exponential behavior, because two circuits are analyzed simultaneously, which leads to internal cancellations. It remains an open question whether efficient verification of arithmetic circuits in the ring $\mathbb{Z}_2[X]$ is possible.

3.2 Incremental Algorithm

In our incremental algorithm, presented in Paper A we partition the multiplier into column-wise slices. The reason to work on columns instead of rows is that the incremental specifications can only be uniquely generated for a column-wise partition. The partial products of a multiplier are uniquely assigned to columns, i.e., all partial products, which have equal coefficients in \mathcal{L} , belong to the same column. But partial products are not uniquely assigned to rows, because they can be permuted arbitrarily within a column without affecting the correctness of a multiplier.

Example 3.3. Fig. 3.1 shows binary multiplication of $13 \cdot 15 = 195$, as it is usually calculated on paper. The order of the partial products differs, depending whether the multiplication started with the most or least significant bit of the right bit-vector. In both multipliers we marked the same partial product. It can be seen that it is contained in different rows, but remains in the same column.

$$\begin{array}{r}
 \begin{array}{ccccccc}
 1 & 1 & 0 & 1 & \cdot & 1 & 1 & 1 & 1 \\
 \hline
 & & & & & 1 & 1 & 0 & 1 \\
 & & & & & 1 & \color{red}{1} & 0 & 1 \\
 & & & 1 & 1 & 0 & 1 & & \\
 & & 1 & 1 & 0 & 1 & & & \\
 \hline
 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1
 \end{array}
 &
 \begin{array}{r}
 \begin{array}{ccccccc}
 1 & 1 & 0 & 1 & \cdot & 1 & 1 & 1 & 1 \\
 \hline
 & & & 1 & 1 & 0 & 1 & & \\
 & & & 1 & 1 & 0 & 1 & & \\
 & & & & 1 & \color{red}{1} & 0 & 1 & \\
 & & & & 1 & 1 & 0 & 1 & \\
 \hline
 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1
 \end{array}
 \end{array}$$

Figure 3.1: Multiplication of $13 \cdot 15 = 195$ with different orderings of partial products.

Even without assuming that partial products are allowed to be permuted in multipliers, it is not possible to derive unique row-wise specifications for all multiplier architectures.

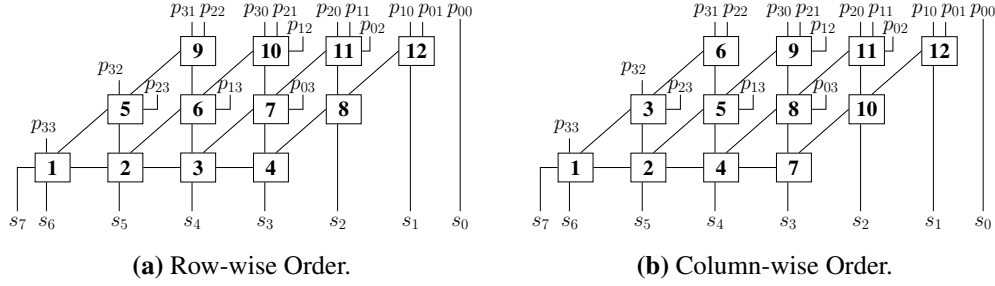


Figure 3.2: Different orders of full- and half-adders, with $p_{ij} = a_i b_j$.

Consider for example the “btor”- and “sp-ar-rc”-multipliers of Fig. 2.2. It can be seen that the columns of both multipliers always contain the same partial products, but the partial products in the rows differ. It is only possible to define incremental row-wise specifications for individual architectures, such as “btor”-multipliers [24].

Furthermore, it is unclear how to define rows in more complex architectures, which cannot be fully decomposed into full- and half-adders. On the other hand, defining column-wise slices is easy. For each output bit s_i , we define the corresponding input cone and slices are derived as the difference of consecutive cones.

3.3 Reduction Ordering

We discuss in Paper A that we fix a reverse topological term ordering on the set of terms to automatically derive a Gröbner basis. The reduction order of the gate polynomials should follow a reverse topological ordering too. This ensures that after a polynomial is used for reduction it never has to be considered again [103]. Reverse topological term orderings are not unique and we have to determine a “good” reduction order to keep the size of the intermediate reduction results small.

However, a private discussion of A. Biere and A. Mishchenko lead to the conjecture that the choice of the reverse topological reduction order does not make a difference, as long as the full- and half-adders in the multipliers are identified and the internal variables of these adders are eliminated before reduction is applied. We presented a (non-published) paper at the student forum of FMCAD 2019, where we elaborated on this conjecture. We summarize the results in this section. In the following experiments we do not apply our incremental algorithm, but reduce the complete word-level specification after preprocessing the gate constraints using our tool AMULET, presented in Paper C.

Given the shape of multipliers, two orderings seem natural, namely a column-wise and a row-wise ordering. Both orderings are shown in Fig. 3.2 for a simple 4-bit “sp-ar-rc” multiplier. The multipliers in Fig. 3.2 have been preprocessed, such that only full- and half-adders remain. The reduction orderings are not limited to these two cases, circuits support various arbitrary reverse topological orderings, e.g., in Fig. 3.2a the ordering $1 > 2 > 5 > 3 > 4 > 6 > 7 > 9 > 8 > 11 > 10 > 12$ is also reverse topological.

In our experiments we compare row-wise and column-wise reduction orderings

to arbitrary reverse topological reduction orderings on two different multipliers. We perform preprocessing, such that the internal variables of the full- and half-adders are eliminated and measure the size and time of the final reduction process.

In our first experiment we select a simple 32-bit “sp-ar-rc”-multiplier. The 4-bit version of this architecture is depicted in Fig. 3.2. The results can be seen in Figs. 3.3 and 3.4. Figure 3.3 shows the time (in seconds) needed to verify the multiplier and the maximum size of the intermediate reduction results of a row-wise (blue) and column-wise (orange) order and 500 arbitrary reverse topological orderings (green). Additionally we list the size and time of our incremental column-wise (red) approach.

It can be seen that the non-incremental approaches are in the same order of magnitude, i.e., the sizes span a range of around 30 monomials. However the incremental column-wise approach produces by far the smallest and fastest result, because it never considers the complete global specification. Figure 3.4 shows the development of the size of the intermediate results for a complete reduction run. We only show the results of one of the 500 arbitrary orderings we considered. Again the non-incremental orders behave similarly, but are out-rivaled by the incremental column-wise approach.

In our second experiment we consider the more complex 32-bit multiplier “bp-wt-rc” that uses Booth-encoding for generating the partial products and where the full- and half-adders are arranged in a Wallace-tree structure. Wallace-tree multipliers are faster than simple carry-save-adder multipliers, but the arrangement of the full- and half-adders is more involved. The results of this experiment can be seen in Fig. 3.5 and 3.6.

In contrast to the “sp-ar-rc”-multiplier, there is a gap of around 300 monomials between the column-wise and row-wise ordering. The sizes of the arbitrary reverse topological orderings are in between. Only the column-wise order has a linearly decreasing trend during reduction. Again our incremental approach outperforms the non-incremental approaches. Interestingly in both experiments the row-wise ordering caused the biggest intermediate results.

Our experiments show that the effect of reduction order highly depends on the circuit architecture. For simple architectures there is almost no difference between the various non-incremental orderings. On the other hand, the reduction order has a tremendous impact for complex multipliers. Interestingly in both experiments the column-wise ordering is very stable, and especially for complex multipliers, there is a clear trend to select a column-wise ordering for polynomial reduction. However, all non-incremental approaches are outperformed by our incremental column-wise approach. Thus these experiments further support our observation that an incremental approach, where the specification is divided into multiple polynomials helps to speed up reduction time.

3.4 Computer Algebra Systems

We used existing CAS to reduce the specification by the gate constraints in Paper A. Our presented tool AIGMULTOPOLY translates multipliers given as AIGs into polynomials and generates code, which can be executed by either Mathematica or Singular, where the polynomial reduction is performed.

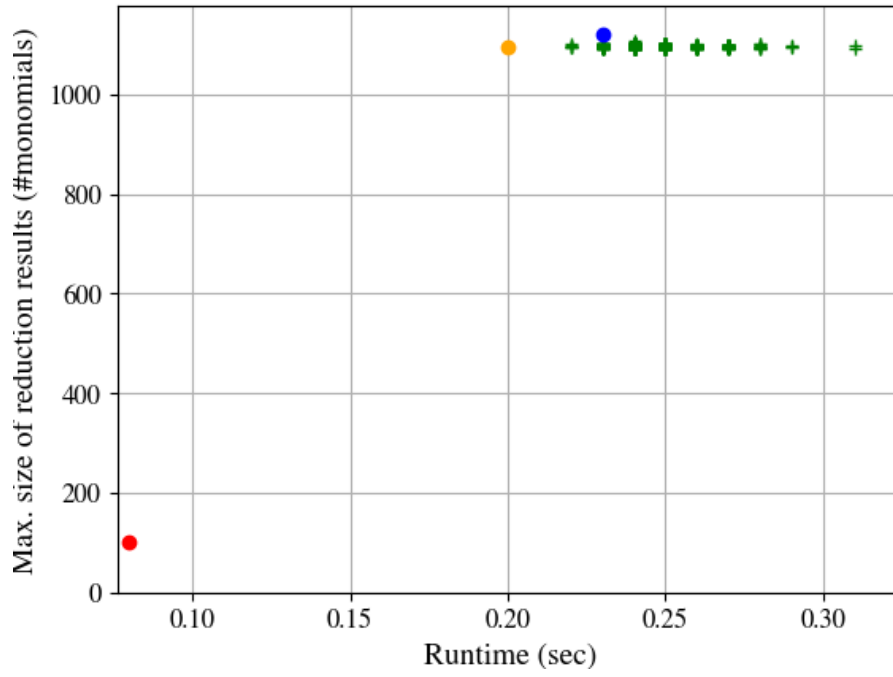


Figure 3.3: Maximum size of reduction results in 32-bit “sp-ar-rc”-multipliers.

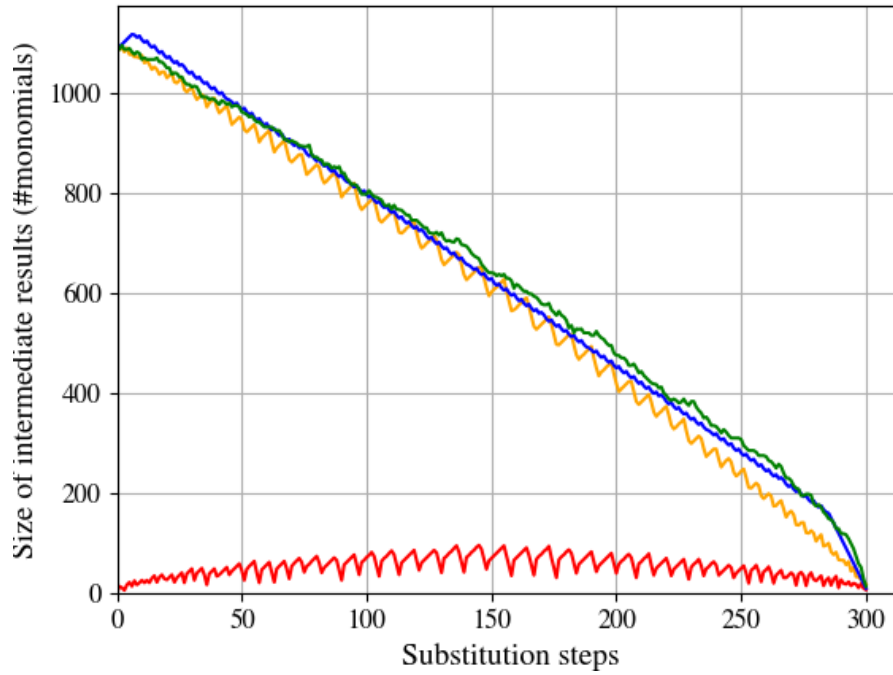


Figure 3.4: Size of reduction results in 32-bit “sp-ar-rc”-multipliers.

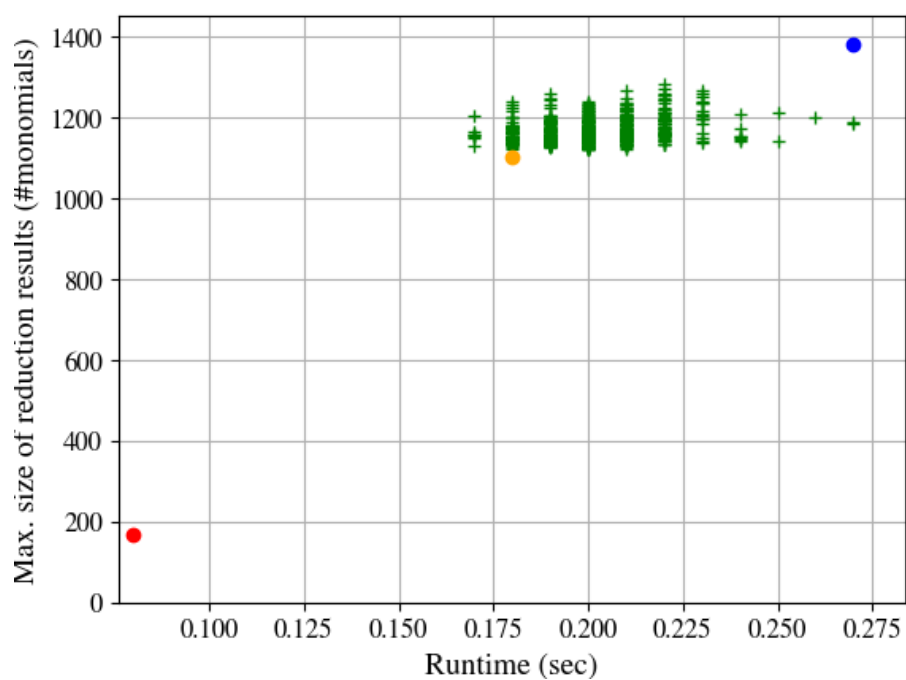


Figure 3.5: Maximum size of reduction results in 32-bit “bp-wt-rc”-multipliers.

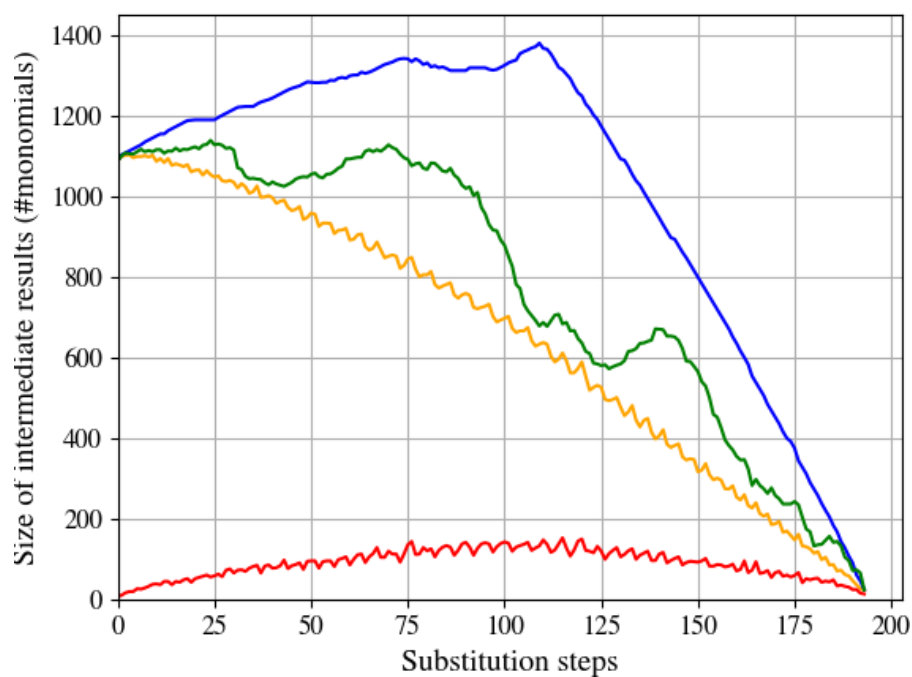


Figure 3.6: Size of reduction results in 32-bit “bp-wt-rc”-multipliers.

Using CAS allowed us a simpler start, where we could focus on modeling and preprocessing, rather than on the implementation of a reduction algorithm. However, CAS are designed for general purposes, which made it hard to encode the verification problem in such a way, that the specific structure of the given set of polynomials is used to full capacity during polynomial reduction. We encountered several problems, which we will discuss in the remainder of this section.

In both systems, Mathematica and Singular, the variable ordering $<$, a set of polynomials $G \subseteq \mathbb{Q}[X]$ and a polynomial $q \in \mathbb{Q}[X]$ need to be provided. Both systems include functions, which calculate whether $q \in \langle G \rangle$, provided G is a Gröbner basis w.r.t. the given order $<$.

Singular treats the given polynomials G as a set, which is internally ordered according to the given variable order. On the other hand, it seems that Mathematica actually treats the set of polynomials as a list. Therefore, it is necessary to provide the polynomials in the desired order. We did not realize this fact in our initial work [89], where we actually printed the polynomials in reverse order. We started by printing the polynomials defining the partial products and ended by printing the polynomial representation of the output bit of each slice. By adjusting the printing order of the polynomials such that the leading terms of the polynomials are ordered according to the given variable order, we were able to improve our computation results of [89] by a factor 2 in Paper A.

Since the reduction order of the polynomials is also fixed by the variable order $<$, it is a big issue to apply polynomial reduction by the Boolean value constraints. To keep the size of the intermediate reduction results small, we desire to immediately eliminate all exponents greater than one. In a CAS, this means that after reduction by a gate constraint, we always apply polynomial reduction by all Boolean value constraints. However, this proved to be practically infeasible.

In Paper A we take a compromise and reduce by the Boolean value constraint of the output variable of a gate, before reduction by the corresponding gate constraint is applied. This turned out to be faster than always reducing by all Boolean value constraints, but has the effect that exponents greater than one will be carried along.

A further issue occurred only in Singular. This CAS introduces a strict limit of 32 767 ring variables, which are already exceeded with 64-bit multipliers. Additionally, as can be seen later in Paper B and as we will discuss in Chap. 4 generating proof certificates in CAS is very slow. All of this issues motivated us to implement our own reduction engine AMULET in Paper C, which is tailored towards reduction of our polynomials and using the full potential of the Boolean value constraints.

Chapter 4

Paper B: A Practical Polynomial Calculus for Arithmetic Circuit Verification

In Paper A we formalized the approach of arithmetic circuit verification using computer algebra and presented how this solving technique can be implemented using existing CAS. However, the verification process might not be error free and thus returns wrong results. In order to increase the trust in reasoning tools, it is common to provide proof certificates, which can be checked by standalone proof checkers to validate the verification results. For algebraic reasoning we require a proof system, which is able to reason about polynomial equations. In Paper B we define a proof calculus, called practical algebraic calculus (PAC), which is able to capture low-level algebraic proofs. Proofs in this format can be checked efficiently.

Let \mathbb{K} be a field and $X = \{x_1, \dots, x_m\}$ be a finite set of variables and further let $f \in \mathbb{K}[X]$ and $G \subseteq \mathbb{K}[X]$. The question we want to answer is, whether the zeroness of G implies the zeroness of f , i.e., every model of G is also a model of f . In algebraic terms this means to derive whether $f \in \langle G \rangle$. In our application G represents the set of gate constraints and Boolean value constraints, and further $\mathbb{K} = \mathbb{Q}$ and $f = \mathcal{L}$. A proof in the PAC format is a sequence of proof rules of the following form:

Addition	$+$	$g_i, g_j, q;$	g_i, g_j appearing earlier in the proof or are contained in G and $q = g_i + g_j$
Multiplication	$*$	$g_i, p, q;$	g_i appearing earlier in the proof or is contained in G and $p \in \mathbb{K}[X]$ being arbitrary and $q = pg_i$

These rules model the properties of an ideal, as defined in Sect. 2.2. Consequently, we can derive for each proof rule that $q \in \langle G \rangle$. In particular if it holds for one rule that $q = f$, we derive that $f \in \langle G \rangle$.

In Paper B, we further show how proofs in this format can be generated by Mathematica in order to provide proof certificates of arithmetic circuit verification. These proofs can either be checked by Singular or by our standalone proof checker, called PACTRIM.

4.1 Comparison to Polynomial Calculus

PAC is highly related to the polynomial calculus (PC) [34], which is introduced in Sect. 2.5.1. PC has been mostly used in proof complexity to obtain lower bounds for the degree and size of proofs, e.g., [55, 75, 82]. Practical applications have not been well-studied. We show in Paper B that we can instantiate the abstract PC in order to gain a proof format, which can be applied in practice.

A major drawback of PC, as it is introduced in [34], is that PC proofs cannot be checked efficiently. A PC proof is only a sequence of polynomials $R = (r_1, \dots, r_n)$. Thus it cannot automatically be determined how a certain polynomial r_i was derived. In order to check a PC proof, we would need to show for each polynomial r_i that it is contained in $\langle r_1, \dots, r_{i-1} \rangle$, which triggers an ideal membership test for each proof rule.

In contrast to PC, we include in PAC the antecedent polynomials g_i , g_j , and p in the proof rules and accordingly keep the information how the conclusion polynomial q was derived. Hence, checking the correctness of PAC proofs can be applied on the fly. For each rule we check that the antecedents are already known (*connection property*) and that the conclusion q is computed correctly (*inference property*).

A further difference of PC and PAC can be seen in the multiplication rules. PC only allows multiplication with terms, whereas PAC permits multiplication by polynomials, which allows shorter proofs. Contrarily PC is less restrictive in the addition rules, where the summands can be multiplied by constants too.

Another significant difference is that PC includes a Boolean axiom encoding that variables can only take values in $\{0, 1\}$. Adding this axiom has the effect that exponents greater than one will be eliminated. However, this restricts models in PC to only be Boolean models. In PAC we support models of $\mathbb{K}[X]$ and thus we do not include these axioms. If we want to encode that certain variables can only take values in $\{0, 1\}$, we have to add the corresponding Boolean value constraints to the set of given polynomials G . We discuss in the following section that for our purpose of arithmetic circuit verification, adding Boolean axioms actually helps to reduce the proof length (number of generated proof rules) and proof size (total number of monomials in the conclusion polynomials).

4.2 Boolean Value Constraints

We already discussed in the last section that we have to explicitly add the Boolean value constraints for each variable to the given set of polynomials G in PAC to introduce Boolean models.

Theoretically, when modeling the circuit, we can only assume that the input variables of multipliers are binary, we cannot automatically derive from the circuit that all internal variables are binary too. Consequently, we would only be allowed to add the Boolean value constraints for the circuit inputs to the given set of polynomials and need to derive the Boolean value constraints for all variables that represent internal AIG nodes.

As can be seen in Ex. 4.1, deriving the Boolean value constraints for each AIG node produces a lot of overhead.

Example 4.1. Let g be the output of an AIG node with two inputs a, b . From $-a^2 + a = 0$, $-b^2 + b = 0$ and $-g + ab = 0$ we can derive that $-g^2 + g = 0$ using the following PAC rules:

$$\begin{array}{llll}
 * : & -g + ab, & g + ab - 1, & -g^2 + g + a^2b^2 - ab; \\
 * : & -a^2 + a, & b^2, & -a^2b^2 + ab^2; \\
 * : & -b^2 + b, & a, & -ab^2 + ab; \\
 + : & -g^2 + g + a^2b^2 - ab, & -a^2b^2 + ab^2, & -g^2 + g + ab^2 - ab; \\
 + : & -g^2 + g + ab^2 - ab, & -ab^2 + ab, & -g^2 + g;
 \end{array}$$

Furthermore, nodes in an AIG that have correct syntax can only represent logical conjunction and whenever the inputs a, b of AIG nodes are binary, the output g of the node has to be binary too. Hence, instead of deriving this information each time, we take the freedom to assume that all variables $x \in X$ can only take values in $\{0, 1\}$ and we add the Boolean value constraints $B(X) = \{x^2 - x \mid x \in X\}$ to the constraint set.

The core of a PAC proof is the number of polynomials of the given set of polynomials G that is used in the proof rules. As can be seen in the experiments of Paper B, the percentage of the proof core is always around 60%. In our experiments we assume correct multipliers that do not include redundant gates, thus all gate constraints have to be used to derive the specification. Hence, the unused polynomials of G have to be Boolean value constraints and the reason for the small core is that we always add all Boolean value constraints to the given set of polynomials G .

The small core and the fact that we assume binary variables in our application, lead to the decision in Paper C that we enable implicit handling of the Boolean value constraints in PAC. We modify PACTRIM in such a way that exponents greater than one are automatically eliminated. For example the following rule is valid:

$$* : x, \quad x, \quad x;$$

The effect of assuming Boolean models can be seen in Table 4.1 and Table 4.2, where we repeat the experiments of Paper B. In Table 4.1 we generate proof certificates, which validate the correctness of “btor” and “sp-ar-rc” multipliers. In the first block “Paper B”, we generate proof certificates as described in Paper B, where we explicitly include the Boolean value constraints. We list the time needed for proof checking (s), the proof length (len), proof size (size) and the degree (d).

In the second block “Paper B + Bool. Model” we rewrite the proof certificates of block “Paper B” and automatically normalize exponents. It can be seen that the proof length decreases by $\sim 40\%$ and the proof size decreases by 30% to 50%, depending on the multiplier. The degree is smaller too. In the third block we present the corresponding proof that is generated by our tool AMULET, which is introduced in Paper C.

Table 4.2 shows the same results for equivalence checking of multipliers. However, it does not contain a block “Paper C”, because equivalence checking is not supported in

mult	n	Paper B				Paper B + Bool. Model				Paper C			
		s	len	size	d	s	len	size	d	s	len	size	d
sparrc	4	0	764	8 156	8	0	542	4 701	5	0	382	2 281	4
sparrc	8	0	3 964	59 330	8	0	2 854	32 813	5	0	1 918	12 817	4
sparrc	16	1	17 804	317 874	8	0	12 950	174 301	5	0	8 446	59 189	4
sparrc	32	3	75 244	1 492 082	8	2	55 030	830 621	5	0	35 326	255 428	4
sparrc	64	15	309 164	6 727 026	8	8	226 742	3 894 685	5	1	144 382	1 071 354	4
btor	4	0	594	4 001	5	0	406	2 632	4	0	268	1 614	3
btor	8	0	2 914	21 915	5	0	1 962	14 314	4	0	1 304	8 715	3
btor	16	0	12 738	104 351	5	0	8 530	69 598	4	0	5 680	40 241	3
btor	32	1	53 122	487 911	5	1	35 490	340 102	4	0	23 648	175 082	3
btor	64	5	216 834	2 387 831	5	3	144 706	1 778 902	4	1	96 448	740 605	3

Table 4.1: Word-level proof checking.

mult	n	Paper B				Paper B + Bool. Model			
		s	len	size	d	s	len	size	d
btor-btor	4	0	1 170	7 952	5	0	802	5 230	4
btor-btor	8	0	5 794	43 902	5	0	3 906	28 732	4
btor-btor	16	0	25 410	210 666	5	0	17 026	141 224	4
btor-btor	32	2	106 114	995 330	5	1	70 914	699 840	4
btor-btor	64	9	433 410	4 942 642	5	6	289 282	3 725 040	4
btor-sparrc	4	0	1 340	12 107	8	0	938	7 299	5
btor-sparrc	8	0	6 844	81 317	8	0	4 798	47 231	5
btor-sparrc	16	1	30 476	424 189	8	0	21 446	245 927	5
btor-sparrc	32	4	128 236	1 999 501	8	2	90 454	1 190 359	5
sparrc-sparrc	4	0	1 510	16 270	8	0	1 074	9 376	5
sparrc-sparrc	8	0	7 894	118 820	8	0	5 690	65 818	5
sparrc-sparrc	16	1	35 542	638 248	8	1	25 866	351 166	5
sparrc-sparrc	32	7	150 358	3 006 256	8	3	109 994	1 683 462	5

Table 4.2: Equivalence proof checking.

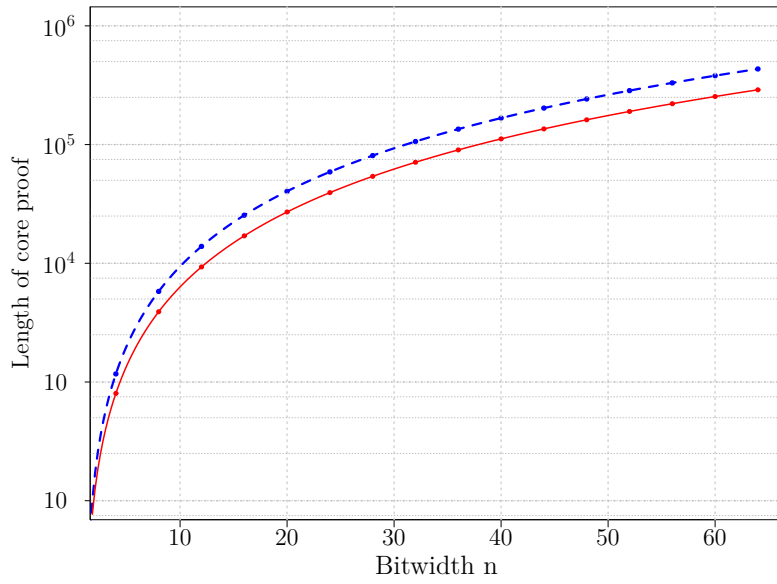


Figure 4.1: Comparison of proof length of “btor-btor” commutativity check with original data (blue, dash) and proofs without exponents (red, solid).

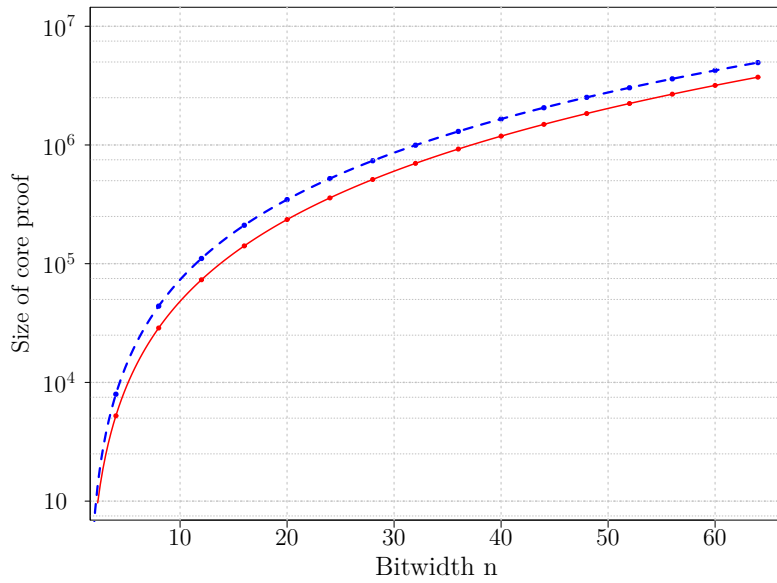


Figure 4.2: Comparison of proof size of “btor-btor” commutativity check with original data (blue, dash) and proofs without exponents (red, solid).

AMULET. Figures 4.1 and 4.2 extend Fig. B.5 of Paper B by adding the proof length and size of the proof certificates, where Boolean models are assumed.

A further effect of assuming Boolean models, despite shorter proofs, is that we do not need the Rabinowitsch trick [35] to derive a complete calculus. We discuss in Paper B, that we need to add a further *radical rule*, which allows us to reason about radical ideals, in order to gain completeness of PAC. In algebra, the radical ideal $\sqrt{\langle G \rangle}$ is defined as

$$\sqrt{\langle G \rangle} = \{p \in \mathbb{K}[X] \mid \exists \alpha \in \mathbb{N} : p^\alpha \in \langle G \rangle\}$$

However, the Rabinowitsch trick, which says

$$f \in \sqrt{\langle G \rangle} \Leftrightarrow 1 \in \langle G \cup \{1 - yf\} \rangle,$$

allows that we do not have to include a radical rule in PAC. Instead we add the polynomial $1 - yf$ with $y \notin X$ to the set of given polynomials G and aim to derive a correct refutation. That is, we need to derive the constant polynomial 1 in the PAC proof.

Usually, we would derive the polynomial f first and add two further rules in order to derive a refutation.

$$\begin{array}{l} \vdots \\ \dots : \quad \dots, \quad \dots, \quad f; \\ * : \quad \quad f, \quad y, \quad yf; \\ + : \quad 1 - yf, \quad yf, \quad 1; \end{array}$$

Since we assume Boolean models in Paper C, we gain soundness and completeness of PAC without using the Rabinowitsch trick and thus can directly stop whenever we derived f . Furthermore, because of Thm. C.11 and Thm. C.15 the soundness and completeness arguments can be generalized to polynomial rings over commutative rings with unity.

4.3 Computer Algebra Systems

We used the CAS Mathematica and Singular as reduction engines in Paper A. Both Mathematica and Singular provide additional information during reduction that allows generating a PAC proof using CAS in Paper B.

We decided to only use Mathematica for generating the proofs, since Singular has a limit on the number of variables. In Paper B we generate proofs in Mathematica, which can be checked either by Singular or by our standalone proof checker PACTRIM. We favor proof checking in PACTRIM, because Singular is orders of magnitude slower than using PACTRIM.

In Mathematica, we use “PolynomialReduce” to generate the proof certificates, which has the following functionality [102]:

`PolynomialReduce[poly, {poly1, poly2, ...}, {x1, x2, ...}]`
yields a list representing a reduction of poly in terms of the poly_i . The list has the form $\{\{a_1, a_2, \dots\}, b\}$, where $a_1 \text{poly}_1 + a_2 \text{poly}_2 + \dots + b = \text{poly}$ and b is minimal.

For correct multipliers it holds that $b = 0$. We generate PAC proofs as follows:

$$\begin{array}{llll}
 * : & \text{poly}_1, & a_1, & a_1 \text{poly}_1; \\
 * : & \text{poly}_2, & a_2, & a_2 \text{poly}_2; \\
 & \vdots & & \\
 + : & a_1 \text{poly}_1, & a_2 \text{poly}_2, & q_1; \\
 + : & q_1, & a_3 \text{poly}_3, & q_2; \\
 & \vdots & & \\
 + : & q_k, & a_m \text{poly}_m, & \text{poly};
 \end{array}$$

It can be seen that the proof certificate of Mathematica, i.e., the list $\{a_1, a_2, \dots\}$ is actually very similar to a Nullstellensatz (NS) proof, cf. Sect. 2.5.2.

However, we apply preprocessing and rewrite the original gate constraints before reducing the specification. For each polynomial that is rewritten in the preprocessing step, we use `PolynomialReduce` to derive a proof certificate, which we translate into PAC. After preprocessing we reduce the (incremental) specification by the simplified Gröbner basis and derive proof certificates in terms of the rewritten polynomials. In order to derive a complete NS proof, we would need to express the rewritten polynomials as linear combinations of the given polynomials, which would increase the proof size.

Generating proof certificates in Mathematica has a very poor performance. For 64-bit multipliers that can be verified within 2 minutes, the proof generation needs more than 6 hours. Furthermore, it is not possible to assume Boolean models in Mathematica. Thus, it is required to include the Boolean value constraints in the given set of polynomials.

The limits of the CAS on the verification itself, as discussed in Chapter 3, and on proof generation were a big motivation to develop our own reduction engine `AMULET` that is presented in Paper C. `AMULET` considers Boolean models and generates proof certificates on the fly, i.e., whenever the specification is reduced by a gate constraint, the corresponding proof rules are generated.

Chapter 5

Paper C: Verifying Large Multipliers by Combining SAT and Computer Algebra

In Paper C, we generalize the algebraic approach of Paper A to be applicable in more general polynomial rings, which admit modular reasoning. In Paper A, we fixed the coefficient domain to \mathbb{Q} , in order to use Gröbner bases theory over fields. We already discussed in Chap. 3 that modeling the circuits in the polynomial ring $\mathbb{Q}[X]$ allows verification of simple multiplier architectures, but fails for more complex architectures, where for example Booth encoding is used. Thus, we generalize our approach and model circuits in polynomial rings $\mathbb{Z}_l[X]$ with $l \in \mathbb{N}$. Furthermore, modular reasoning enables verification of truncated multipliers.

As a consequence, we are not able to use Gröbner bases theory over fields. However, we show in Paper C that we can convert the ideal membership problem in $\mathbb{Z}_l[X]$ to an ideal membership problem in $\mathbb{Z}[X]$, and thus can apply D-Gröbner bases theory [9, 76], which requires the coefficient domain to be a principal ideal domain (PID).

Additionally, we generalize the preprocessing techniques of Paper A, where we applied syntactic pattern matching to identify XOR-gates, full- and half-adders, and nodes with only one parent in the circuit. Only small syntactic changes in the circuits make the preprocessing techniques of Paper A fail to recognize the pattern. In Paper C, we present a preprocessing technique that applies more general rules for variable elimination and subsumes the methods “XOR-rewriting”, “Adder-Rewriting”, and “Common-Rewriting” of Paper A. We will further discuss variable elimination in Sect. 5.1.

Nonetheless, parts of the multiplier, i.e., the final-stage adders (FSA) are hard to verify using computer algebra. Certain adders, namely so-called generate-and-propagate (GP) adders, lead to an exponential blow-up in the intermediate reduction results.

In a GP adder with inputs $x_0, \dots, x_m, y_0, \dots, y_m, c_{\text{in}}$ and outputs $s'_0, \dots, s'_m, c_{\text{out}}$ 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 $c_{m+1} = c_{\text{out}}$ and $c_0 = c_{\text{in}}$. 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, which are not shared internally. Rewriting these sequences leads to exponentially sized polynomials, as can be seen in the following example.

Example 5.1. Let $o = o_1 \vee x_0$, $o_2 = o_3 \vee x_1$, $o_3 = x_2 \vee x_3$ represent a sequence of three OR-gates, which can be combined to $o = x_0 \vee x_1 \vee x_2 \vee x_3$. The corresponding polynomial representation in $\mathbb{Z}[o, x_0, x_1, x_2, x_3]$ is $o = x_0 + x_1 - x_0x_1 + x_2 - x_0x_2 - x_1x_2 + x_0x_1x_2 + x_3 - x_0x_3 - x_1x_3 + x_0x_1x_3 - x_2x_3 + x_0x_2x_3 + x_1x_2x_3 - x_0x_1x_2x_3$ and contains $2^4 - 1$ monomials.

Our first attempt to overcome this issue, was to preprocess complex FSA in such a way that the sequences of OR-gates are shared internally. We were successful for multipliers where the FSA is a carry look-ahead adder. However, this approach relied on syntactic pattern matching and we would have to define a new rewriting strategy for each individual adder architecture.

While preparing benchmarks for the SAT Race 2019 [67], we noticed that checking the equivalence of adder circuits is trivial for SAT solvers. Based on this observation we combine SAT and computer algebra in our verification technique. When the FSA is syntactically identified as a GP adder, we replace it by a simple ripple-carry adder. A bit-level miter is generated and is given to SAT solvers in order to prove the equivalence of the adders. The rewritten multiplier is verified using computer algebra.

We implement our own dedicated reduction engine *AMULET* in C, which is able to apply adder substitution and verifies the (rewritten) multipliers. The polynomial reduction algorithm in *AMULET* makes use of the specific shape of the circuit polynomials, i.e., all leading coefficients of the gate constraints are -1 and the leading terms contain only one unique variable. In Paper C we call this property *UMLT property*. Consequently, polynomial reduction reduces to substitution of every occurrence of the leading variable by the tail of the polynomial.

Furthermore, we handle the Boolean value constraints implicitly, i.e., we immediately normalize exponents greater than one. As already discussed, CAS handle this reduction explicitly. Using the UMLT property and immediate reduction by Boolean value constraints makes *AMULET* more time efficient than CAS, which can be seen in the experiments of Paper C and in Chap. 9.

5.1 Variable Elimination

The rewriting techniques presented in Paper A and Paper C eliminate variables from the set of gate constraints in order to rewrite and thus simplify the generated Gröbner basis.

First, we have to select a variable v , which represents an internal node in the AIG, as elimination variable. All polynomials in the (D-)Gröbner basis are reduced by the corresponding gate constraint with leading term v . This has the effect that only this gate constraint will contain v and we eliminate this polynomial from the (D-)Gröbner basis. We provide correctness theorems in Paper A for Gröbner bases and in Paper C for D-Gröbner bases that prove that we are allowed to locally rewrite the (D-)Gröbner basis and the result will again be a (D-)Gröbner basis for the elimination ideal.

The question is which variables v shall be eliminated from the set of gate constraints in order to simplify reduction. In Paper A we apply syntactic pattern matching to select the elimination variables v . First, we apply “Adder-Rewriting”, where we search

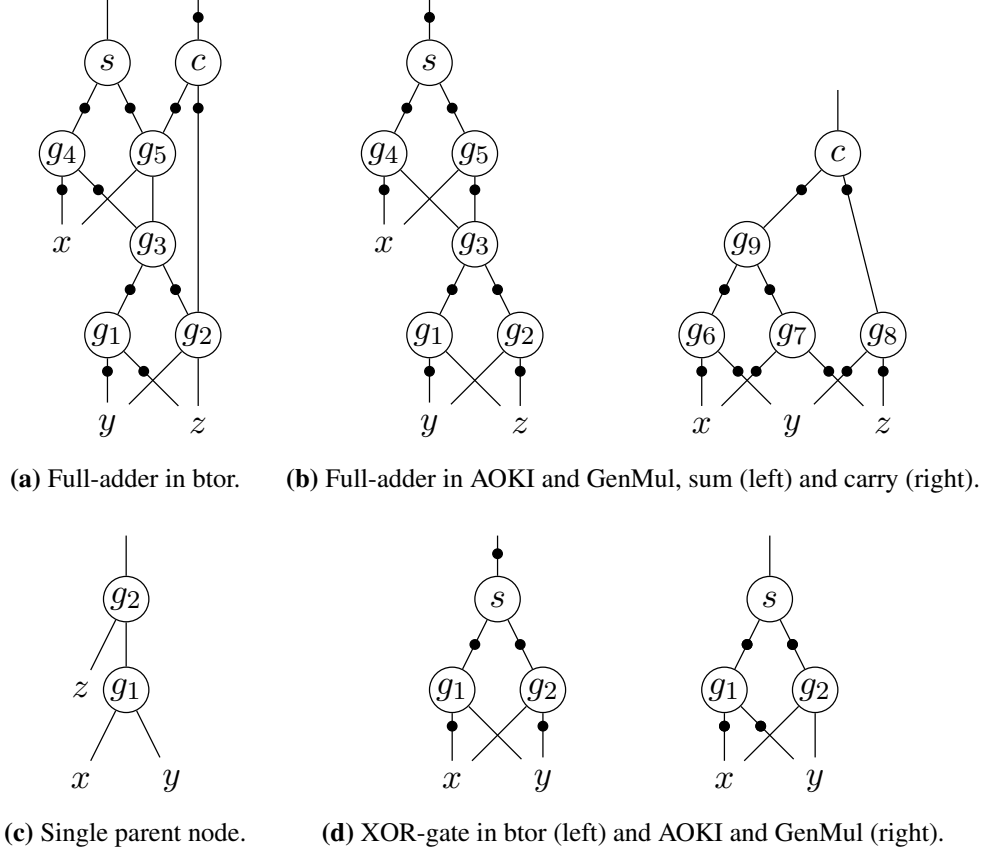


Figure 5.1: Patterns considered in the rewriting technique of Paper A.

for sub-circuits representing full- and half-adders in the multipliers and eliminate the corresponding internal variables. This has the effect that the linear adder specifications, e.g., $-2c - s + x + y + z$ for full-adders, are included in the Gröbner basis. Subsequently, we apply “XOR-Rewriting” and search for sub-circuits representing XOR-gates and eliminate the internal variables. In our last rewriting technique “Common-Rewriting” we identify nodes in the AIG that only have one parent node and eliminate them. However, all our techniques are highly tailored to the selected benchmarks, generated by Boolector (btor) [83], AOKI [53], and GenMul [81]. The covered rewriting patterns, except for half-adders, can be seen in Fig. 5.1. If only small changes in the circuits, e.g., bit-flipping or adding redundant nodes are applied, the rewriting approach fails.

In Paper C we propose a more general approach for variable selection that does not rely on specific patterns in the AIG. In contrast to Paper A, we select the variables directly in the D-Gröbner basis, after the column-wise slices are fixed. We search for polynomials p in the sliced D-Gröbner bases, whose leading variable only occurs in the tail of one other polynomial q that has to be contained in the same slice. The remainder of reducing q by p is added to the D-Gröbner basis and both polynomials p and q are removed from the D-Gröbner basis.

Our approach is related to “Common-Rewriting” of Paper A. However, in “Common-Rewriting” we iterate only once over the AIG to identify nodes that have only one parent. In our new approach we iterate multiple times over the D-Gröbner basis until a fix-point has been reached and all leading variables either occur in multiple polynomials or in other slices too.

It can easily be seen in Figs. 5.1c and 5.1d that our approach of Paper C fully subsumes “Common-Rewriting” and “XOR-Rewriting” of Paper A.

For “Adder-Rewriting”, this is not entirely true and depends on the selected benchmarks. For the AOKI and GenMul benchmarks, depicted in Fig. 5.1b the variables $g_1, g_2, g_4, g_5, g_6, g_7, g_8, g_9$ are eliminated in the first iteration over the D-Gröbner basis. The node g_3 will not be eliminated in the first iteration, because it occurs in the gate constraints of g_4 and g_5 . However, in the second iteration only the gate constraint with leading term s contains the variable g_3 . Thus g_3 will be eliminated in the second iteration and hence all internal variables are eliminated and two polynomials with leading terms s and c remain.

In “Adder-Rewriting” of Paper A we merge these two Polynomials to derive the linear polynomial $-2c - s + x + y + z$ and further keep the polynomial with leading term s to maintain completeness. We do not generate the adder specification in Paper C, because we do not want to generate a polynomial with leading coefficient -2 . Since the approach of Paper C is defined over \mathbb{Z} , division by 2 is not possible in general.

Furthermore, we define the reduction order in AMULET in such a way that the carry polynomial and the sum polynomial will always be considered for reduction directly after another and thus the blow-up in the intermediate reduction results is very limited.

For “btor”-multipliers more polynomials remain when the full-adder is rewritten, as only g_1, g_4 are eliminated in the first iteration and g_3 is eliminated in the second iteration. However, the remaining polynomials with leading terms s, c, g_5, g_2 are subsequently considered for reduction.

5.2 Proof Certificates

AMULET generates PAC proofs as by-product of the polynomial reduction. The precise algorithms for generating the proof certificates are presented in Paper D, where we give a comprehensive tool description of AMULET.

Let G be the set of gate constraints and $B(X)$ be the set of Boolean value constraints. The syntax of the PAC rules is basically the same as presented in Paper B, except that we add a deletion information in the prefix of each rule. Furthermore, we assume Boolean models and thus automatically reduce exponents greater than one, which is indicated by the side condition “ q being reduced by $B(X)$ ”.

$$\begin{array}{ll} \text{Addition} & d + : g_i, \quad g_j, \quad q; \quad \begin{array}{l} g_i, g_j \text{ appearing earlier in the proof} \\ \text{or are contained in } G \text{ and} \\ q = g_i + g_j \text{ being reduced by } B(X) \end{array} \end{array}$$

Multiplication $d * : g_i, p, q;$ g_i appearing earlier in the proof
or is contained in G
and $p \in \mathbb{Z}[X]$ being arbitrary and
 $q = pg_i$ being reduced by $B(X)$

Adding the deletion information is similar to clause deletion for SAT proofs [49] and helps to reduce the memory usage of our proof checker PACTRIM. Initially the constraint set in PACTRIM consists of the given polynomials G . While checking the proof rules for correctness, the conclusion polynomial q of each rule is added to the constraint set. Thus, the constraint set increases with every proof rule, which leads to memory exhaustion for large proof certificates.

We add a deletion information in the prefix of a proof rule whenever both antecedents g_i and g_j of the current rule do not occur in subsequent proof rules anymore. This has the effect that g_i and g_j will be removed from the constraint set, which decreases the memory usage. However, adding a deletion information always affects both antecedents of a proof rule. In Paper F we add standalone deletion rules that allow us to delete individual polynomials from the constraint set. We provide experiments in Paper F that show the positive effect of the deletion rules on the memory usage. Furthermore, we extend PAC by adding extension rules and introduce indices to name polynomials.

In the experiments of Paper C, we generate and check DRUP and PAC proofs for 64-bit multipliers, cf. Table C.1. We did not generate PAC proofs for multipliers with an input bit-width larger than 64, which are presented in Table C.2.

We close this gap and these experiments are now included in Table 5.1 and Table 5.2 and follow the same structure as the experiments presented in Paper C.

Times are listed in seconds for Table 5.1 and in minutes for Table 5.2. In both tables we first list the multiplier architecture and the input bit-width n . In the block “Verify” we show the time needed for adder substitution (“sub”) and the time CADICAL needs for equivalence checking (“cnf”). Column “aig” shows the time AMULET needs to verify the rewritten multiplier. The computation times are summed up in column “tot”.

The second block “Certify” shows the time of the same tools when proof generation is enabled. Proof generation does neither affect adder substitution nor the SAT solver, but has an effect on the verification time of AMULET. The block “Check” shows the time DRAT-TRIM [99] needs to validate the DRUP-proof generated by the SAT solver (“cnf”) and the time PACTRIM needs to validate the PAC proof (“aig”). Note, that we exclude the column “cnf” from Table 5.2, because the corresponding computation times are always less than 30 seconds, and thus all entries in the respective column are zero. Furthermore, we list the input size of the AIG, i.e., the number of nodes and the proof length of the DRUP and PAC proofs.

architecture	n	Verify				Certify				Check			total	input	proof len	
		sub	cnf	aig	tot	sub	cnf	aig	tot	cnf	aig	tot		aig	cnf	aig
btor	128	0	0	9	10	0	0	15	15	0	10	10	26	0.1	0	0.5
kjvkv	128	0	0	9	9	0	0	14	14	0	15	15	29	0.2	0	0.8
sp-ar-rc	128	0	0	10	10	0	0	14	14	0	16	16	30	0.2	0	0.8
sp-dt-lf	128	0	2	13	15	0	2	19	21	1	17	18	40	0.2	0.1	0.8
sp-wt-bk	128	0	1	18	20	0	1	20	21	0	18	18	40	0.2	0.1	0.8
btor	256	1	0	119	120	1	0	160	160	0	60	61	222	0.5	0	2.0
kjvkv	256	1	0	84	86	1	0	113	113	0	87	87	201	0.8	0	3.1
sp-ar-rc	256	1	0	84	86	1	0	114	114	0	92	92	207	0.8	0	3.1
sp-dt-lf	256	3	6	164	174	3	6	169	176	2	102	104	283	0.8	0.4	3.1
sp-wt-bk	256	3	3	170	177	3	3	176	180	1	105	106	289	0.8	0.3	3.1

Table 5.1: Certification Time (time in sec, input and proof length in 10^6).

architecture	n	Verify			Certify			Check		total	input	proof len	
		sub	aig	tot	sub	aig	tot	aig	tot		aig	cnf	aig
btor	512	0	16	16	0	23	23	7	7	30	2.1	0	7.8
kjvkv	512	0	13	13	0	15	15	9	9	25	3.1	0	12.3
sp-ar-rc	512	0	13	13	0	16	16	10	10	26	3.1	0	12.3
sp-dt-lf	512	1	25	26	1	25	26	11	11	37	3.1	1.0	12.3
sp-wt-bk	512	1	26	27	0	26	26	11	11	38	3.2	0.6	12.4
btor	1024	2	177	179	2	219	219	51	51	272	8.4	0	31.4
kjvkv	1024	2	91	93	2	172	172	72	72	245	12.6	0	49.2
btor	2048	17	1493	1510	17	2552	2552	430	430	2982	33.5	0	125.8
kjvkv	2048	18	1129	1147	18	2077	2077	1228	1228	3307	50.3	0	197.0

Table 5.2: Certification Time (time in min, input and proof length in 10^6).

Chapter 6

Paper D: SAT, Computer Algebra, Multipliers

Paper D is an extension of Paper C. In Paper C we combined SAT and computer algebra to tackle verification of complex multiplier architectures. In our approach we identify whether the multiplier circuit contains a complex FSA and, if necessary, substitute the complex adder by a simpler adder circuit. The replacement is verified using a SAT solver and the rewritten multiplier is verified using computer algebra techniques.

In Paper C we introduced our tool AMULET that applies adder substitution and verification of integer multipliers fully automatically. We briefly described AMULET in Paper C, however, the main focus in Paper C is on the theoretical aspects of combining SAT and computer algebra.

In Paper D we give a rigorous system description of AMULET and present the different options “substitution”, “verify”, and “certify”. In the “substitution”-phase, we identify whether the FSA is a GP adder thanks to the specific structure that distinguishes them from simple adders. If the FSA is a GP adder, we replace it by an equivalent ripple-carry adder. A bit-level miter is generated in order to verify the equivalence of the two adder circuits and a rewritten multiplier circuit in AIG format is returned.

We always apply “substitution” before the input multiplier is verified or certified. In “verify”, AMULET reads the (rewritten) multiplier, applies the preprocessing steps as presented in Paper C, and verifies the circuit using our incremental verification algorithm presented in Paper A. The incremental verification algorithm of Paper A is tailored towards the multiplier specification of unsigned integer multipliers. However, we also generalized our incremental verification algorithm to signed and truncated multipliers in Paper C. At that point we did not present how the incremental algorithm of Paper A can be generalized to different multiplier specifications. We close this gap in Paper D, where we present small adjustments that are necessary to make our algorithm applicable for signed and truncated multipliers too.

The “certify”-phase extends “verify” and a proof certificate in the PAC format is generated, which can be checked by our standalone proof checker PACTRIM. We treated proof generation only on a high level in Paper C. In Paper D we discuss how the proof rules are generated in AMULET and present the differences of proof generation during preprocessing and when the incremental verification algorithm is applied.

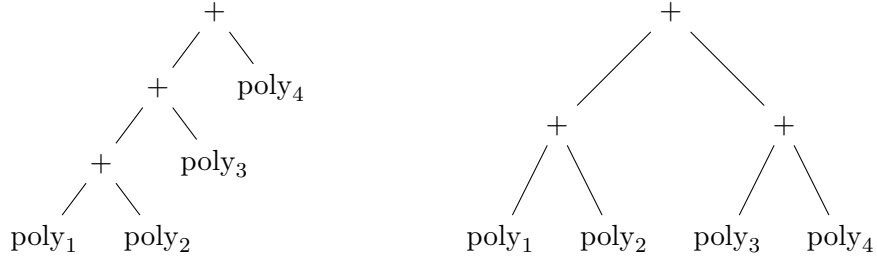


Figure 6.1: Addition of 4 polynomials in sequence (left) and tree-based (right).

6.1 Proof Generation

In the preprocessing steps we eliminate variables from the set of gate constraints to rewrite and simplify the D-Gröbner basis. In order to eliminate a variable v from the D-Gröbner basis, we reduce a polynomial p , which contains v in the tail, by the gate constraint q , whose leading term is equal to v . Since both polynomials p and q are contained in the ideal, we generate a multiplication and addition rule to cover the rewriting step.

Example 6.1. Let $p = 5x - 2y$ and $q = -y + 3z$ such that $I = \langle p, q \rangle \subseteq \mathbb{Z}[x, y, z]$ and assume we want to eliminate y from I . Reduction of p by q yields the polynomial $r = 5x - 6z \in \mathbb{Z}[x, z]$. The corresponding PAC rules are

$$\begin{aligned} * : & \quad -y + 3z, & -2, & \quad 2y - 6z; \\ + : & \quad 5x - 2y, & 2y - 6z, & \quad 5x - 6z; \end{aligned}$$

In the incremental verification algorithm, we reduce the column-wise specifications by the elements of the sliced D-Gröbner basis. In contrast to preprocessing, the column-wise specifications are not contained in the D-Gröbner basis. If we would construct the proof as in Ex. 6.1, checking the addition rule would yield an error, because the column-wise specifications neither appear earlier in the proof nor are contained in the set of given polynomials.

To overcome this issue we generate proofs similar to the approach in Paper B. We generate the multiplication rules and push each conclusion polynomial of a multiplication rule on a stack. When reduction of a slice-wise specification is completed, we generate addition rules, which sum up the elements of the stack in order to derive the column-wise specifications. Finally, the column-wise specifications are multiplied by constants and are summed up to yield the full circuit specification \mathcal{L} .

We do not sum up the polynomials in sequence, but use a tree-based addition approach, cf. Fig. 6.1 in order to reduce the size of the intermediate polynomials.

Example 6.2. Let $\mathcal{L} = -8s_3 - 4s_2 - 2s_1 - s_0 + 4a_1b_1 + 2a_0b_1 + 2a_1b_0 + a_0b_0$ be the specification of a 2-bit multiplier and further let $I = \langle -s_3 + c_1, -s_2 - 2c_1 + c_0 + a_1b_1, -s_1 - 2c_0 + a_1b_0 + a_0b_1, -s_0 + a_0b_0 \rangle \subseteq \mathbb{Z}[s_3, s_2, s_1, s_0, c_1, c_0, a_1, a_0, b_1, b_0]$ be the ideal generated by the column-wise specifications.

mult	n	tree		sequence		factor
		time	size	time	size	
sp-ar-rc	4	0.01	1 845	0.01	1 942	1.05
sp-ar-rc	8	0.02	10 537	0.01	11 938	1.13
sp-ar-rc	16	0.06	48 997	0.06	63 518	1.29
sp-ar-rc	32	0.28	212 516	0.34	346 774	1.63
sp-ar-rc	64	1.60	895 418	2.28	2 063 750	2.30
btor	4	0.01	1 291	0.01	1 362	1.05
btor	8	0.01	7 112	0.01	8 800	1.23
btor	16	0.05	33 198	0.05	54 252	1.63
btor	32	0.24	145 639	0.33	353 476	2.42
btor	64	1.54	620 282	2.70	2 473 332	3.98

Table 6.1: Proof size and checking time (in sec) for different addition approaches.

We show that $\mathcal{L} \in I$ and the corresponding PAC rules are

$$\begin{aligned}
& * : -s_3 + c_1, 8, -8s_3 + 8c_1; \\
& * : -s_2 - 2c_1 + c_0 + a_1b_1, 4, -4s_2 - 8c_1 + 4c_0 + 4a_1b_1; \\
& * : -s_1 - 2c_0 + a_1b_0 + a_0b_1, 2, -2s_1 - 4c_0 + 2a_1b_0 + 2a_0b_1; \\
& + : -8s_3 + 8c_1, -4s_2 - 8c_1 + 4c_0 + 4a_1b_1, -8s_3 - 4s_2 + 4c_0 + 4a_1b_1; \\
& + : -2s_1 - 4c_0 + 2a_1b_0 + 2a_0b_1, -s_0 + a_0b_0, -2s_1 - s_0 - 4c_0 + 2a_1b_0 + 2a_0b_1 + a_0b_0; \\
& + : -8s_3 - 4s_2 + 4c_0 + 4a_1b_1, -2s_1 - s_0 - 4c_0 + 2a_1b_0 + 2a_0b_1 + a_0b_0, \mathcal{L};
\end{aligned}$$

This example is very small, thus adding up the factors in sequence does not make a difference in regard of the number of monomials in the conclusion polynomials:

$$\begin{aligned}
& + : -8s_3 + 8c_1, -4s_2 - 8c_1 + 4c_0 + 4a_1b_1, -8s_3 - 4s_2 + 4c_0 + 4a_1b_1; \\
& + : -8s_3 - 4s_2 + 4c_0 + 4a_1b_1, -2s_1 - 4c_0 + 2a_1b_0 + 2a_0b_1, \\
& \quad - 8s_3 - 4s_2 - 2s_1 + 4a_1b_1 + 2a_1b_0 + 2a_0b_1; \\
& + : -8s_3 - 4s_2 - 2s_1 + 4a_1b_1 + 2a_1b_0 + 2a_0b_1, -s_0 + a_0b_0, \mathcal{L};
\end{aligned}$$

However, this example still shows the idea of our tree-based approach, which is to avoid that we collect and carry along the n^2 partial products. We run experiments on “btor”- and “sp-ar-rc”-multipliers for different input bit-widths n and either add the polynomials in sequence or in a tree-based scheme. The resulting proof sizes and checking times are depicted in Table 6.1. It can be seen that adding the polynomials in sequence leads to larger proof sizes and thus longer checking time and the factor increases with the bit-width. For 64-bit “btor”-multipliers the proof size is four times larger when the polynomials are added in sequence compared to a tree-based scheme.

6.2 Proof Size

We elaborate on the proof metrics of “btor”-multipliers in Paper D. As already discussed, this multiplier architecture is very simple and consists only of full- and half-adders, which are arranged in a grid-like structure, cf. Fig. 2.2.

In Paper B we empirically examine proof length, size, and degree of these benchmarks and derive a proof length in $\mathcal{O}(n^2)$, proof size in $\mathcal{O}(n^3)$ and a degree of 5. In Table 4.1 of Chap. 4, we give experimental evidence that the degree, proof length, and proof size of the certificates generated by AMULET are smaller compared to the proof certificates generated by the CAS Mathematica.

We derive formal proofs of these metrics in Paper D. We give a precise polynomial bound of $16n^2 - 20n - 1$ for the proof length and prove that the degree is 3. We are only able to derive an upper bound of $\mathcal{O}(n^2 \log(n))$ for the proof size because of the tree-based addition approach. Depending on the number of polynomials that are summed up, different intermediate summands are generated.

In Paper D we measure degree, proof length, and size only for the addition and multiplication rules. We do not consider the original constraint set in the proofs of Thms. D.20, D.21, and D.22. In the following corollaries we adapt the proof metrics to also contain the set of given polynomials, i.e., the gate constraints $G(C) \cup \{l\}$. It can be seen that the degree and the complexity of the proof sizes remain the same and only the polynomial bound for the proof length changes.

Corollary 6.3. *The degree of PAC proofs of “btor”-multipliers, including the set of given polynomials, is 3.*

Proof. We need to investigate whether $G(C) \cup \{l\}$ contains polynomials with a degree higher than 3. The set $G(C)$ contains only polynomials induced by AIG nodes. These polynomials have degree 2, as can be seen in Fig. D.4. Since l has degree 0, the highest degree in the PAC proof remains 3. \square

Corollary 6.4. *The proof length of “btor”-multipliers, including the set of given polynomials, is $24n^2 - 9n$.*

Proof. We need to add the number of polynomials in $G(C) \cup \{l\}$ to the polynomial bound $16n^2 - 20n - 1$, which is derived in Thm. D.20. Following from Lemma D.19 $|G(C)| = 8n^2 - 9n$ and thus the proof length is $16n^2 - 20n - 1 + 8n^2 - 9n + 1 = 24n^2 - 9n$. \square

Corollary 6.5. *The proof size of “btor”-multipliers, including the set of given polynomials, is in $\mathcal{O}(n^2 \log(n))$.*

Proof. The size of each polynomial in $G(C)$ is at most 5, see Fig. D.4. Thus the size of $G(C) \cup \{l\}$ is at most $5(8n^2 - 9n) + 1$, which does not affect the complexity class. \square

Chapter 7

Paper E: From DRUP to PAC and Back

In Paper C we combined SAT and computer algebra to successfully verify large complex multiplier architectures. If the FSA in a multiplier is a GP adder, we replace the complex adder by an equivalent ripple-carry adder. A bit-level miter is generated, which is translated into CNF and given to a SAT solver that verifies the correctness of the substitution step. The rewritten multiplier is verified by our tool AMULET using computer algebra techniques.

In both tools we generate proof certificates, which validate the verification results. However, since two different verification techniques are involved, two proof certificates in different formats are generated. The SAT solver generates proofs in DRUP [49] format, which can be validated by DRAT-TRIM [99] and AMULET generates PAC proofs, which can be checked by our proof checker PACTRIM.

The proofs are not connected, which leaves a hole in the certification argument. It would be possible to apply compositional reasoning using interactive theorem proving [54], but that requires manual interaction.

In Paper E we show how to merge the two proof certificates in order to derive a single proof certificate in one format. We show how DRUP proofs can be translated into PAC proofs and vice versa. To translate the DRUP proof generated by the SAT solver into PAC, we encode the CNF as a set of polynomials and generate PAC rules that model the resolution steps used to derive the RUP clauses. Both PAC proofs are merged in order to derive one single PAC proof.

To get from PAC rules to DRUP, we generate a SMT encoding of the PAC proof. SMT solvers are able to translate SMT encodings into AIGs, which can be further converted into CNF. The generated CNF is given to a SAT solver, where a DRUP proof is generated. The DRUP proofs are combined in order to derive one single proof.

We demonstrate the abilities of translating one proof format into the other on our use case of Paper C. However, the demonstrated techniques are not limited to this application. It turns out that PAC proofs are complete, more compact and faster to check. Due to the usage of SMT solvers gaps remain in the merged DRUP proof.

7.1 DRUP to PAC

AMULET generates the bit-level miter as an AIG, which is then internally converted into CNF. We convert DRUP proofs into the PAC format by first encoding each node of the AIG as a polynomial. These polynomials are added to the original constraint set of the PAC proof. Algebraic operations are applied to derive the polynomial representation of the clauses of the corresponding CNF, cf. Ex. E.3. Second, we need to convert the derived RUP clauses into PAC. We generate PAC rules, which model the resolution steps needed to derive each RUP clause.

7.1.1 Traces

A DRUP proof only lists the RUP clauses and does not contain information how these clauses are derived. Hence we decided in Paper E, to use the *TraceCheck* proof format instead. The TraceCheck format is a compact proof format for resolution proofs and has the format “idx clause 0 antecedents 0”, where the antecedents are the indices of the clauses used to derive the conclusion clause using resolution. Lines with trailing double zeros mark initial clauses.

Example 7.1, taken from Paper E, shows a DRUP proof and a TraceCheck proof for the same SAT problem. It can be seen that the derived clauses in the DRUP format and the TraceCheck format are the same. However, the TraceCheck format, contains the information we need to generate PAC rules.

Example 7.1. This is an unsatisfiable CNF in DIMACS format (left) with a DRUP (middle) and a TraceCheck (right) proof.

p	cnf	3	5		-2	0		1	1	-2	-3	0	0		
1	-2	-3	0	d	3	0		2	1	2	0	0			
1	2		0	d	1	-2	-3	0	3	-1	-2	0	0		
-1	-2		0	d	-1	-2	0		4	-1	2	0	0		
-1	2		0		0				5	3	0	0			
		3	0						6	-2	0	3	1	5	0
									7	0	4	2	6	0	

In Paper C we use the SAT solver CADICAL [15] to verify the bit-level miter. However, CADICAL only produces DRUP proofs and does not provide TraceCheck proof certificates. Thus we use the SAT solver PICOSAT [11] in Paper E.

The size of the corresponding PAC proof depends highly on the proof generated by the SAT solver and it frequently happens that parts of the traces are equal. In Paper E we do not identify whether sub-traces are shared and repeatedly generate the same proof rules, which adds redundancy but reduces the amount of polynomials that the checker has to consider at a given point of time, because we can apply deletion.

The experiments in Table 7.1 depict the amount of redundancy in our proofs. We use the same benchmarks as in Paper E and show the time and memory needed to check the PAC proof certificate. Furthermore, we list the number of proof rules and the percentage of redundant proof rules. In column “max” we list the maximum number of repetitions for one single proof line.

architecture	n	Paper E					No redundancy		
		sec	MB	length	red	max	sec	MB	length
sp-ar-cl	8	0	11	37 167	36%	35	0	8	19 230
sp-bd-ks	8	0	14	57 079	36%	53	0	14	31 175
sp-dt-lf	8	0	15	53 850	36%	47	0	13	28 464
bp-ct-bk	8	0	11	46 115	34%	42	0	9	24 976
bp-wt-cl	8	0	17	67 951	35%	37	0	18	37 002
sp-ar-cl	16	2	48	185 588	39%	128	1	47	93 095
sp-bd-ks	16	2	56	209 249	37%	85	1	53	113 016
sp-dt-lf	16	1	36	136 349	34%	45	1	33	76 734
bp-ct-bk	16	1	33	128 720	35%	53	1	29	69 685
bp-wt-cl	16	11	165	614 742	45%	256	9	202	267 246
sp-ar-cl	32	32	405	1 597 897	47%	511	26	568	628 053
sp-bd-ks	32	8	224	817 956	40%	230	6	215	421 014
sp-dt-lf	32	3	82	321 720	33%	69	2	75	184 096
bp-ct-bk	32	2	57	217 128	30%	46	2	50	133 490
bp-wt-cl	32	248	1 716	5 536 176	49%	884	212	3 687	2 096 071
sp-bd-ks	64	18	400	1 440 943	34%	145	15	395	861 254
sp-dt-lf	64	10	206	816 572	32%	95	8	179	479 951
bp-ct-bk	64	7	119	459 262	24%	56	6	107	311 302

Table 7.1: Proof Checking (in bold the fastest/most memory efficient version).

It can be seen that between 30% and 50% of the proof rules are redundant and for example for “sp-ar-cl-32” the same proof rule is derived 511 times. We modify the tool DRUP2PAC of Paper E to detect and remove repeated proof rules. The results are shown in the second block of Table 7.1. It can be seen that proof checking time is around 25% faster and the proof length is significantly lower compared to proofs that include redundant proof rules.

7.1.2 Extensions

To reduce the size of polynomials that model clauses, we add polynomials of the form $-f_x + 1 - x = 0$ to the original constraint set, where x represents a variable that occurs in the PAC proof and f_x has to be a new variable. By adding the above equation we model that f_x represents the negation of the Boolean variable x .

In PCR, cf. Sect. 2.5.1, these polynomials are added as axioms to PC in order to admit shorter polynomial representations of clauses. In PAC we do not automatically assume these polynomials as axioms, but add them to the initial constraint set.

Example 7.2. The clause $x \vee y \vee z$ can be translated to $\bar{x} \wedge \bar{y} \wedge \bar{z} = \perp$ using De Morgan's laws. The corresponding polynomial equation is $(1 - x)(1 - y)(1 - z) = 0$, which generates 2^3 monomials, when expanded. If on the other hand we introduce $-f_x + 1 - x = 0$, $-f_y + 1 - y = 0$, $-f_z + 1 - z = 0$, the same equation can be depicted as $f_x f_y f_z = 0$, consisting of one monomial.

However, extending the original constraint set with additional polynomials can change the models. We do not check the original constraint set, thus it might happen that we add a polynomial, which affects the models of the constraint set. For example, we could simply add the constant polynomial 1 to our knowledge base, which makes any PAC proof obsolete. Thus, we want to assume as few polynomials as possible, i.e., in our application we only want to assume the gate constraints as original constraint set. We address this issue in Paper F, where we add a proper extension rule to PAC.

7.2 PAC to DRUP

To convert PAC proofs into DRUP proofs, we encode each PAC rule in SMT format [5] using the theory over quantifier-free fixed size bit-vectors.

Example 7.3. Consider the following PAC proof. We are given the polynomials $3x - z$ and $2y - 3x$ and generate the following PAC rules

$$\begin{aligned} + : & \quad 3x - z, \quad 2y - 3x, \quad 2y - z; \\ * : & \quad 2y - z, \quad 2, \quad 4y - 2z; \end{aligned}$$

Checking the correctness of this rule can be encoded as:

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 1))
(declare-fun y () (_ BitVec 1))
(declare-fun z () (_ BitVec 1))
(assert
  (let (($v0 (bvadd (bvand #b0011 ((_ sign_extend 3) x))
                    (bvand #b1111 ((_ sign_extend 3) z)))))
    (let (($w0 (bvadd (bvand #b0010 ((_ sign_extend 3) y))
                    (bvand #b1101 ((_ sign_extend 3) x)))))
      (let (($p0 (bvadd (bvand #b0010 ((_ sign_extend 3) y))
                    (bvand #b1111 ((_ sign_extend 3) z)))))
        (let (($e0 (= (bvadd $v0 $w0) $p0)))

          (let (($v1 (bvadd (bvand #b0010 ((_ sign_extend 3) y))
                    (bvand #b1111 ((_ sign_extend 3) z)))))
            (let (($w1 #b0010))
              (let (($p1 (bvadd (bvand #b0100 ((_ sign_extend 3) y))
                    (bvand #b1110 ((_ sign_extend 3) z)))))
                (let (($e1 (= (bvmul $v1 $w1) $p1)))

                  (not (and $e0 $e1))))))))))
(check-sat)
```

For each PAC rule we define corresponding formulas v_i , w_i , p_i , and e_i . In a correct PAC proof all e_i are true, thus the SMT formula is unsatisfiable. We use the SMT solver BOOLECTOR [83] to solve the SMT formula. Furthermore BOOLECTOR is able to generate a corresponding AIG, which is translated into CNF using the tool AIG2CNF from the AIGER library [16].

We state in Paper E, that this encoding leaves gaps in the certification arguments. First of all, our modeling is not complete. For correct PAC proofs, three properties need to hold. The first property is the *inference property*, i.e., that the conclusion polynomials of each rule are computed correctly. The second property is the *connection property*, which checks that the antecedent polynomials of the rules are already known. The third property is that the target polynomial is inferred or that a correct refutation was derived. The SMT encoding of Paper E only checks the inference property, which can also be seen in Ex. 7.3. In order to check the connection property, we could introduce a naming variable for each original constraint and use these names as antecedents in the proof rules, cf. Ex. 7.4.

Example 7.4. Consider again the PAC proof from Ex. 7.3. We encode the SMT formula as follows:

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 1))
(declare-fun y () (_ BitVec 1))
(declare-fun z () (_ BitVec 1))
(assert
  (let (($g0 (bvadd (bvand #b0011 ((_ sign_extend 3) x))
                    (bvand #b1111 ((_ sign_extend 3) z)))))
    (let (($g1 (bvadd (bvand #b0010 ((_ sign_extend 3) y))
                    (bvand #b1101 ((_ sign_extend 3) x)))))

      (let (($p0 (bvadd (bvand #b0010 ((_ sign_extend 3) y))
                    (bvand #b1111 ((_ sign_extend 3) z)))))
        (let (($e0 (= (bvadd $g0 $g1) $p0)))

          (let (($w1 #b0010))
            (let (($p1 (bvadd (bvand #b0100 ((_ sign_extend 3) y))
                    (bvand #b1110 ((_ sign_extend 3) z)))))
              (let (($e1 (= (bvmul $p0 $w1) $p1)))

                (not (and $e0 $e1))))))))))
(check-sat)
```

Checking that the target polynomial is inferred, is more involved and possibly requires the usage of quantifiers, because we need to model polynomial equality for all possible choices of input variables. This would close the gaps from the modeling side. However, as discussed in Paper E, we are not able to track internals of SMT solving. Thus the generated AIG contains simplifications that are not covered in the proof, which still leaves a gap in the certificate. Closing this gap is an interesting future work.

Chapter 8

Paper F: The Proof Checkers Paccheck and Pastèque for the Practical Algebraic Calculus

In Paper B we introduced the practical algebraic calculus (PAC), which is able to cover proof certificates for algebraic reasoning. We extend this format in Paper F and present our new proof checkers **PACHECK** and **PASTÈQUE**. **PACHECK** is implemented in C and checks PAC proofs in the new syntax very efficiently. **PASTÈQUE** is verified in Isabelle/HOL. As stated in Sect. 1.2, **PACHECK** is implemented by the author of this thesis and **PASTÈQUE** is implemented and verified by one of the co-authors of Paper F.

PAC is based on the polynomial calculus (PC) [34]. PC is well-studied on the theory side but has practical limits that are discussed in Chap. 4. PAC extends PC by adding information of the antecedents, such that efficient proof checking is possible. However, in PAC as defined in Paper B, the antecedent polynomials have to be provided explicitly. To reduce the size of the proof file, we introduce *indices* to name polynomials, and thus derive a more condensed proof format, because we don't spell out the antecedent polynomials again but access them by their names.

Example 8.1. Consider the following PAC proof. The file `<input>` contains the given set of polynomials and the PAC rules are contained in `<proof>`. The left side shows the PAC proof in the format of Paper B, the right side shows the condensed format as introduced in Paper F.

<pre><input> x-y; xz+yz+z; <proof> *: x-y, z, xz-yz; +: xz-yz, xz+yz+z, xz+z;</pre>	<pre><input> 1 x-y; 2 xz+yz+z; <proof> 3 * 1, z, xz-yz; 4 + 3, 2, xz+z;</pre>
---	---

Furthermore, we define a standalone *deletion rule*, which helps to reduce the memory usage of our proof checkers. In Papers C and D we already presented a basic version of a deletion rule, where we added “d” in the prefix of a proof rule to mark that the antecedents of a corresponding rule can be deleted from the constraint set. In Paper F, we introduce a deletion rule, which allows deleting individual rules, similar to clause deletion in DRUP.

multiplier	length (10 ⁶)	deg	PACHECK						PASTÈQUE			
			no delete		no index		default		uloop			
			sec	MB	sec	MB	sec	MB	sec	MB	sec	MB
sparcl-32	1.6	256	23	773	36	354	21	353	121	40 654	113	9 492
spdtlf-32	0.3	46	2	122	3	73	2	73	11	1 679	11	886
bpctbk-32	0.2	25	1	86	2	52	1	51	8	1 600	7	1 068
bpwtcl-32	5.6	764	193	4 324	302	1 430	181	1 428	786	58 867	774	64 404

Table 8.1: Proof Checking (in bold the fastest version).

As a further contribution, we discuss an *extension rule*. In Paper E we presented PAC proofs, where we modified the original constraint set and added polynomials to derive shorter proof rules. In PCR these polynomials are added as axioms [3]. However, as we discussed in Chap. 7, adding new polynomials to the original constraint set may affect the models of the original constraint set and thus introduce errors. Hence, we propose an extension rule that checks that extensions do not change the models of the given set of polynomials. In the experiments of Paper F, we modify the experiments of Paper E and add all polynomials defining Boolean negation as extensions to the PAC proof. We discuss in Sect. 8.1 that we can further enhance the experiments of Paper E, such that only the gate constraints of the input multiplier are contained in the original constraint set. All constraints, which are generated by adder substitution, are added as extensions, which allows cleaner proof certificates.

8.1 Extensions

Recall our verification approach of Paper C, where we substitute complex FSAs in multipliers by simple ripple-carry adders. A bit-level miter is generated, which is passed on to a SAT solver to verify the equivalence of the adders. Computer algebra techniques are used to verify the correctness of the rewritten multiplier, cf. Fig. D.3. Since two different solving techniques are used, two proof certificates in different formats are generated. SAT solvers generate a DRUP proof and computer algebra techniques produce a PAC proof. In the experiments of Paper E, where we translate the DRUP proof into PAC, all gate constraints of the given multiplier, the equivalent ripple-carry adder, and the bit-level miter are assumed as original constraints in the PAC proof. We even added polynomials that define Boolean negation to the original constraint set.

We are able to further modify our experiments and derive PAC proofs, which only contain the gate constraints of the given multiplier as original constraints. All other polynomials, which are induced by the ripple-carry adder and the bit-level miter are added as extension rules to the PAC proof. Thus, the original constraint set of the PAC proof only consists of the gate polynomials induced by the given multiplier. Table 8.1 contains the modified experiments, which will be included in the final version of Paper F.

architecture	length (10 ⁶)	add		add + sort	
		sec	MB	sec	MB
btor-128	0.4	5	90	5	90
btor-256	1.6	24	356	26	356
btor-512	6.3	137	1 420	144	1 421
sparrc-128	0.6	6	134	6	134
sparrc-256	2.3	27	532	28	532
sparrc-512	9.4	132	2 130	140	2 130
sparecl-32	1.6	21	352	21	352
spdtlf-32	0.3	2	73	2	73
bpctbk-32	0.2	1	51	1	51
bpwtcl-32	5.6	181	1 426	181	1 426

Table 8.2: Comparison of “add” and “add+sort” for the performance of PACHECK.

8.2 PACHECK

In Paper F we give a system description of our proof checker PACHECK, which validates PAC proofs in the new syntax and is also backwards compatible to the original syntax of PAC presented in Paper B.

8.2.1 Addition algorithm

In PACHECK we modify the polynomial-addition algorithm that is used in PACTRIM. In our previous proof checker PACTRIM we add two polynomials by pushing the monomials of both polynomials on a stack. The stack is sorted and monomials with equal terms are merged.

However, polynomials are internally stored as ordered linked lists of monomials in PACHECK. Thus, the monomials in the polynomials are already sorted and we use this property while adding two polynomials. We iterate over both polynomials simultaneously and always push the largest monomial on a stack. If both polynomials contain a monomial with the same term, we merge the monomials by adding the coefficients and push the merged monomial on the stack unless it is zero. The monomials on the stack are automatically ordered and we do not have to sort the stack. Table 8.2 shows the effect of the addition algorithm on the benchmarks of Paper F. The columns “add” show the time and memory needed of PACHECK. In the columns “add + sort” PACHECK uses the addition algorithm of PACTRIM, where the stack needs to be sorted. It can be seen that the effect is rather limited and only in proofs with more than a million rules a small optimization of the computation time is visible.



Figure 8.1: Term representation w.r.t. $v > u > x > y$ (left) and $x > u > y > v$ (right).

8.2.2 Sorting

Terms and polynomials are represented as ordered linked lists in PACHECK, which are sorted based on a defined variable order. The terms are shared internally, hence it is important to choose an appropriate order, because the variable order has an influence on the number of generated terms and thus on the memory usage of PACHECK.

Example 8.2. Assume we want to represent the terms uxy and vxy . Figure 8.1 shows the internal representation of these terms for two different variable orderings. For the ordering $v > u > x > y$ depicted on the left side, the internal sharing is maximal and only 4 terms are allocated. For the ordering $x > u > y > v$ depicted on the right side, terms cannot be shared and thus 6 terms need to be allocated.

It is crucial to identify a good term ordering. However, the best ordering that maximizes internal sharing cannot be determined in advance from the original constraint set, as it highly depends on the applied operations in the proof rules.

In PACHECK as presented in Paper F, we order variables using the function “strcmp”, which lexicographically sorts by the names of the variables. A further option is to use the same variable ordering as in the given proof files. That is, whenever we read a new variable from a proof file, we assign an increasing numeric value to the variable and sort according to this value. In Table 8.3 we compare these two orderings and also include the respective reverse orderings. It can be seen that there is no clear preference towards a specific ordering and the number of generated terms highly depends on the given problem. We added the ability to select between the four orderings to PACHECK.

mult	len (10 ⁶)	strcmp		rev. strcmp		level		rev. level	
		terms	MB	terms	MB	terms	MB	terms	MB
btor-128	0.4	421 k	90	453 k	90	421 k	90	453 k	90
btor-256	1.6	1 694 k	356	1 823 k	355	1 695 k	356	1 824 k	356
btor-512	6.3	6 796 k	1 420	7 317 k	1 420	6 797 k	1 421	7 318 k	1 421
sparrc-128	0.6	711 k	134	694 k	133	726 k	134	727 k	134
sparrc-256	2.3	2 864 k	532	2 797 k	532	2 927 k	532	2 928 k	532
sparrc-512	9.4	11 494 k	2 130	11 230 k	2 129	11 752 k	2 130	11 754 k	2 130
sparcl-32	1.6	9 685 k	352	17 931 k	478	17 931 k	478	9 681 k	352
spdtlf-32	0.3	456 k	73	539 k	78	547 k	78	451 k	72
bpctbk-32	0.2	264 k	51	260 k	52	266 k	52	261 k	51
bpwtcl-32	5.6	88 115 k	1 426	151 162 k	2 559	152 788 k	2 567	86 641 k	1 415

Table 8.3: Sorting methods.

Chapter 9

Evaluation

In this chapter we evaluate the contributions of this thesis and compare our introduced approaches to related work. 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 rounded seconds (wall-clock time) and the time limit is set to 300 seconds.

9.1 Circuit Verification

In the experiments in this section we verify multiplier circuits without proof generation. We compare our approaches of Papers A and C to related work [31, 80]. As discussed in Sect. 2.6 the tools of [93] are not available, thus a comparison is not possible.

In the approach of Paper A we use AIGMULTOPOLY to translate multipliers given as AIGs into polynomials. We apply preprocessing techniques, where we eliminate internal variables of certain sub-circuits, which are identified using syntactic pattern matching. The CAS Mathematica [102] and Singular [38] are used to apply polynomial reduction based on our incremental verification algorithm. In the experiments we measure the time from starting AIGMULTOPOLY until Mathematica resp. Singular is finished.

We enhance the method in Paper C, where we combine SAT and computer algebra to identify and replace complex FSA of the multiplier. SAT solvers are used to check the equivalence of the adder circuits and our dedicated reduction engine AMULET verifies the rewritten multiplier using computer algebra techniques. We measure the time AMULET needs to apply adder substitution and circuit verification and include the time the SAT solver CADICAL [15] needs to verify the equivalence of the adders. We compare our tools to the current state-of-the-art of related work, cf. Sect. 2.6:

Mahzoon et al. [80] The authors of [80] propose a rewriting technique, where converging gate cones in multipliers are detected and rewritten. Their tool RevSCA uses so-called “atomic blocks” in order to speed-up rewriting by reducing the search space for finding converging gates. Recently RevSCA-2.0 was published, which improves RevSCA and also supports verification of signed multipliers.

Yu et al. [31] The method of [31] reduces the word-level output of a multiplier by the circuit constraints in order to derive the input signature of the multiplier. The rewriting algorithm has been integrated into the ABC [10] tool. A flag “atree” can be used to

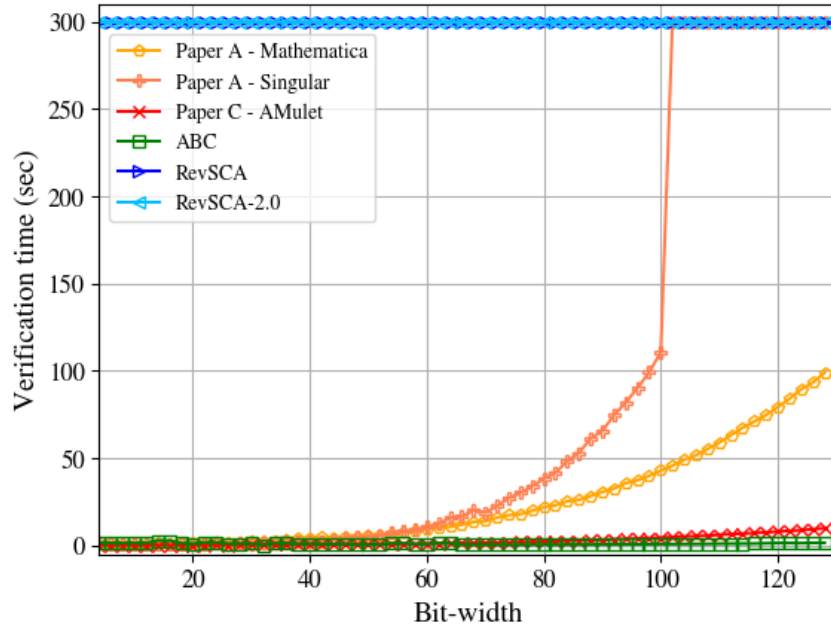


Figure 9.1: Verification time of “btor”-multipliers (in sec).

identify and rewrite full- and half-adders in the multipliers. However, this flag leads to incompleteness, when the multipliers cannot be fully decomposed into adder trees.

9.1.1 Simple Multipliers

In our first experiment, we select the “btor”-multipliers with input bit-width n in the interval $[4, 128]$, generated by Boolector [83]. These multipliers can be fully decomposed into full- and half-adders, as depicted in the left side of Fig. 2.2. The results are shown in Fig. 9.1 and it can be seen that our approach of Paper C is much faster than our initial approaches of Paper A. Singular has the problem that the number of ring variables is already exceeded for input bit-width $n = 100$. The tool of Yu et al. [31], with “atree” enabled, is slightly faster than our approach of Paper C. Both tools of [80] fail to verify “btor”-benchmarks, RevSCA due to a segmentation fault and RevSCA-2.0 due to incompleteness (returns “buggy multiplier”).

We repeat the experiment using “sp-ar-rc”-multipliers with input bit-widths n in the interval $[4, 128]$, which are generated using the tool GenMul [81]. These multipliers can be seen on the right side of Fig. 2.2 and can be fully decomposed into full- and half-adders too. Figure 9.2 shows the experimental results. The results of the approaches of Paper A, Paper C and Yu et al. [31] are very similar to the experiments shown in Fig. 9.1. It can again be seen that the approach of Paper C is much faster than our initial approaches of Paper A, but slightly slower than the method of Yu et al. [31]. The tools of Mahzoon et al. [80] are able to verify “sp-ar-rc”-multipliers, but are slower than our approach of Paper C.

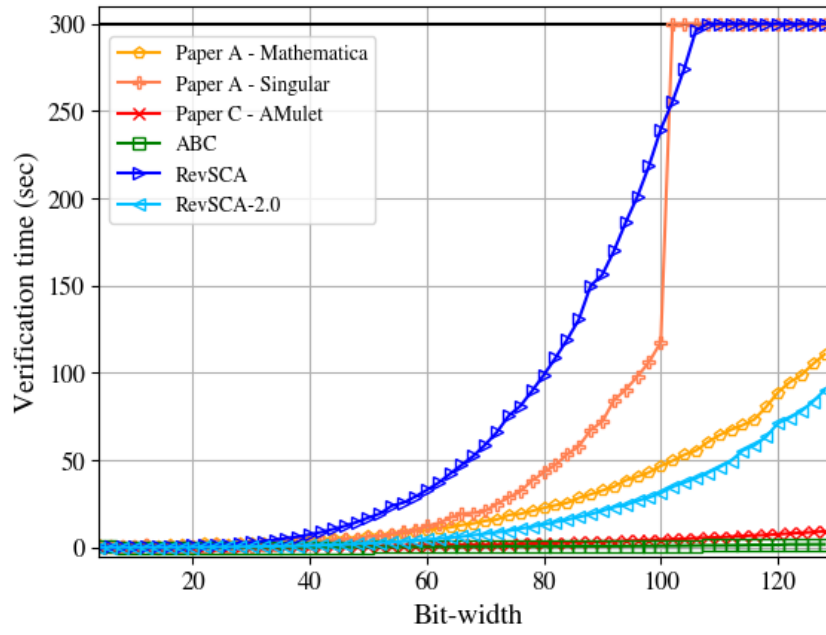


Figure 9.2: Verification time of “sp-ar-rc”-multipliers (in sec).

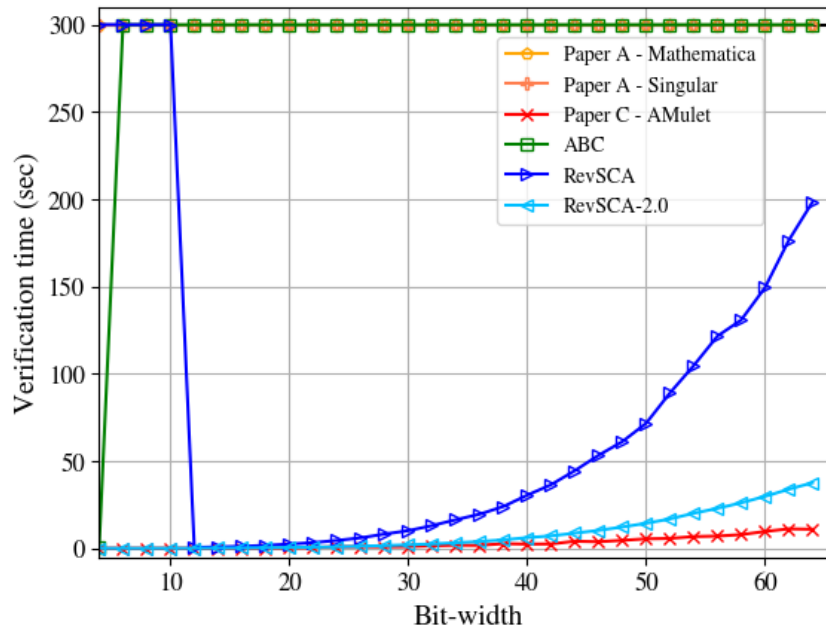


Figure 9.3: Verification time of “bp-wt-cl”-multipliers (in sec).

9.1.2 Complex Multiplier

In these experiments we select “bp-wt-cl”-multipliers, which are part of the AOKI-benchmarks [53]. These multipliers are highly optimized to yield a logarithmic computation time. Partial products are generated using Booth-encoding, which are accumulated in a Wallace-tree structure and the FSA is a carry look-ahead adder. The AOKI-benchmarks are only available up to input bit-width 64, thus we generate benchmarks with input bit-width n in the interval $[4, 64]$.

For ABC [31], we turned off the option “atree”, because it leads to incompleteness. The results can be seen in Fig. 9.3. Both approaches of Paper A and ABC exceed the time limit for very small input bit-widths. RevSCA [80] times out for small benchmarks but is able to verify multipliers of larger input bit-width. Our approach of Paper C is faster than both tools of Mahzoon et al. [80].

9.1.3 Benchmark Suite

In order to derive a more comprehensive comparison over different multiplier architectures, we consider all possible multipliers of the AOKI benchmarks with input bit-width 64. Following components can be combined to gain 192 different multipliers:

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

The results can be seen in Fig. 9.4. Again, we deactivated the option “atree” in ABC. Our approaches of Paper A are able to solve only one benchmark, namely the “sp-ar-rc” multipliers. The approach of Yu et al. [31] is not able to solve any multiplier. Our approach of Paper C is much faster than the solvers RevSCA and RevSCA-2.0 of Mahzoon et al. [80], but we are able to solve fewer instances in total compared to RevSCA-2.0.

We compare AMULET to RevSCA-2.0 in Fig. 9.5 and Table 9.1. Multipliers, which either use a “(7,3) counter tree” or a “redundant binary addition tree” as PPA are colored in red and it can be seen that our approach fails to verify multipliers that contain these architectures. We are currently investigating this issue.

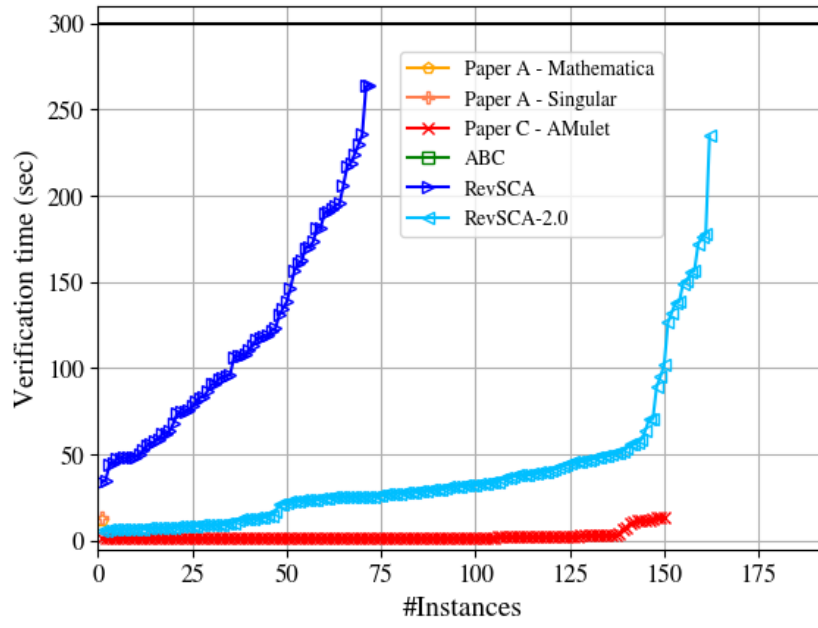


Figure 9.4: Verification of AOKI multipliers (in sec).

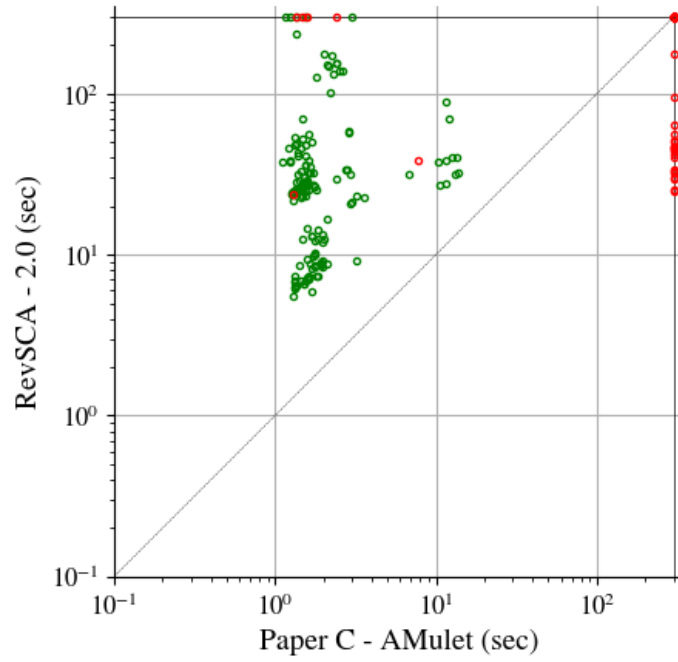


Figure 9.5: Comparison of AMULET and RevSCA-2.0 (in sec).

		AMULET		total
		solved	TO	
RevSCA-2.0	solved	141	21	162
	TO	8	10	18
	buggy	1	11	12
total		150	42	192

Table 9.1: Comparison of AMULET and RevSCA-2.0.

Furthermore, for some instances RevSCA-2.0 returns “buggy multiplier”, although AMULET is able to verify (some of) these multipliers and even provides proof certificates. These multipliers are contained in the line “buggy” in Tbl. 9.1.

9.2 Proof Generation and Checking

In this section we present experiments on proof generation and checking, where we compare the methods presented in Paper B, C, and F. None of the related work supports proof generation. The approach of Paper B uses Mathematica to generate the proof certificates, which are checked by Singular or our own proof checking tool PACTRIM.

In the approach of Paper C, we generate the proof certificates as by-product of verification in our reduction engine AMULET, which are checked by PACTRIM.

We introduce a new syntax of PAC in Paper F, where we add deletion and extension rules and use indices to name polynomials. AMULET is modified to generate certificates in the new syntax, which are checked by PACHECK or PASTÈQUE.

We certify “btor”-benchmarks up to input bit-width 128. The results for generating the certificates can be seen in Fig. 9.6. It can be seen that proof certificates can only be generated for multipliers with input bit-width less than 30 using our approach of Paper B. The time needed to generate the proofs for Paper C and Paper F is the same, because only the syntax is adapted in AMULET and the process for proof generation remains the same.

Figure 9.7 shows that proof checking using our own implemented proof checking tools is much more efficient than using Singular. Furthermore, the fastest proof checker is PACHECK, which uses the condensed PAC syntax, introduced in Paper F.

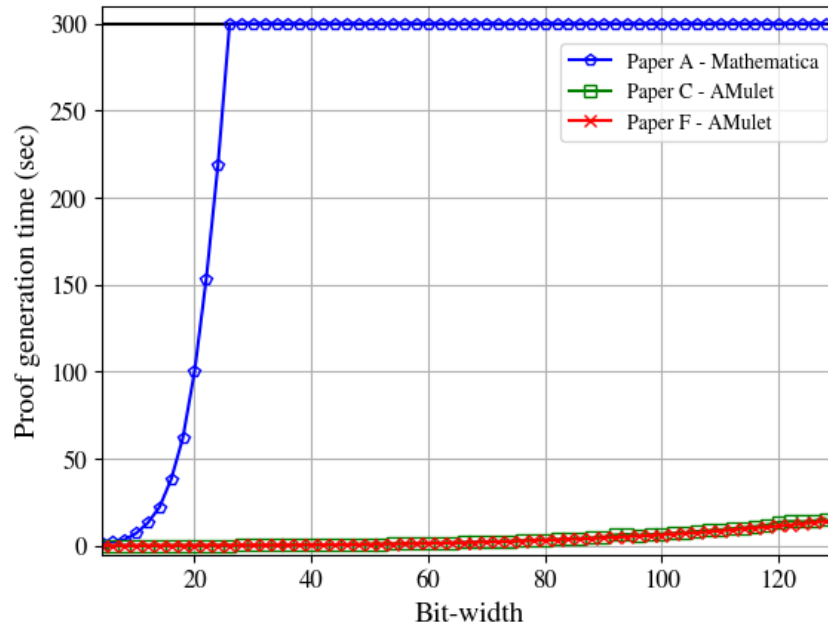


Figure 9.6: Proof generation of “btor”-multipliers (in sec).

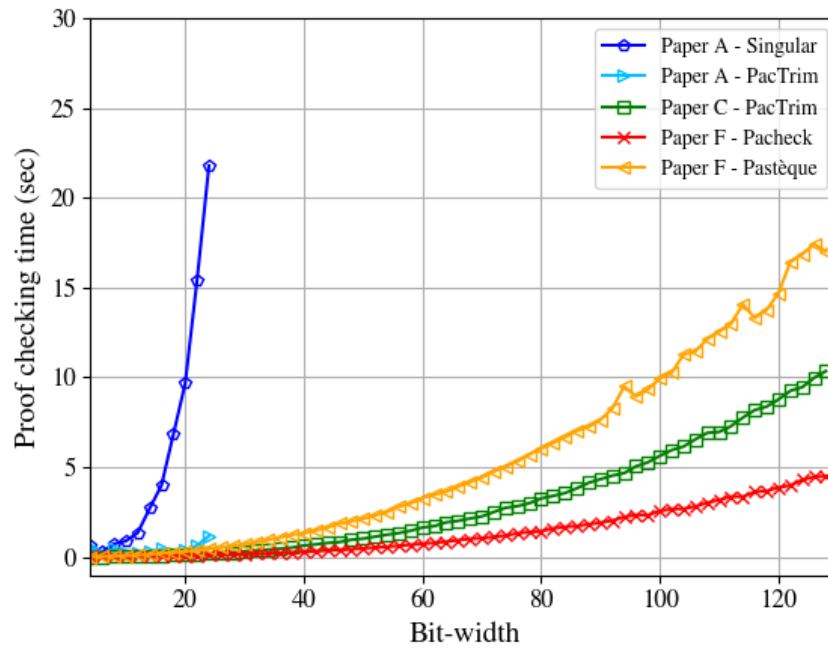


Figure 9.7: Proof checking of “btor”-multipliers (in sec).

Chapter 10

Conclusion

In this thesis, we presented several techniques to verify arithmetic circuits, in particular multiplier circuits, using computer algebra. Furthermore, we developed a practical proof calculus, which allows us to efficiently generate and check proof certificates in order to validate the results of the automated reasoning tools.

The developed approaches were topics of six publications, which are included as Papers A–F in the second part of the thesis. In Part I of the thesis, i.e., Chap. 3–8, we discussed the ideas and results and reflected on the contributions of these papers. Additionally we gave a comprehensive evaluation of our developed techniques in Chap. 9.

In Paper A we gave a rigorous formalization of the approach of arithmetic circuit verification using computer algebra for polynomial rings over fields and proved soundness and completeness. We developed an incremental verification algorithm based on column-wise slicing, which has the effect that the verification problem can be split into smaller, more manageable sub-problems. Furthermore, we proposed rewriting techniques that apply syntactic pattern matching to identify and rewrite structures in the circuits. Our tool AIGMULTOPOLY translates circuits into polynomials, which can be passed on to computer algebra systems, where polynomial reduction is applied. In Chap. 3 we elaborated on the selected polynomial ring and discussed how modeling our problem in polynomial rings over fields leads to bigger intermediate reduction results than for polynomial rings, which admit modular reasoning. We presented why the incremental algorithm can only be applied column-wise and showed the effects of different reduction orderings. Additionally, we discussed the drawbacks of using computer algebra systems for arithmetic circuit verification.

Since the verification approach might not be error-free, we introduced in Paper B a practical proof calculus, which allows us to capture low-level algebraic proofs. Our calculus PAC instantiates the polynomial calculus (PC) [34] and can be checked efficiently by our tool PACTRIM. In Chap. 4, we gave a more comprehensive comparison of the proof formats PAC and PC and furthermore elaborated on the effects of adding Boolean axioms to PAC. We presented in detail the procedure of generating proof certificates in computer algebra systems.

In Paper C, we generalized the approach of multiplier verification to be applicable in more general rings, which allow modular reasoning. We further combine SAT and computer algebra to verify complex multiplier architectures. Certain parts of the multiplier, i.e., the final-stage adders are hard to verify using computer algebra. In our approach we identify whether the final-stage adder is a generate-and-propagate adder

and, if necessary substitute the complex adder by an equivalent ripple-carry adder. The replacement is checked by SAT solvers and the rewritten multiplier is verified using our own implemented reduction engine AMULET. We generalized our preprocessing techniques of Paper A, such that they do not rely on specific patterns in the multiplier. In Chap. 5 we discussed why the new preprocessing approaches are more powerful and subsume the preprocessing techniques of Paper A. Furthermore, we expanded the discussion on proof generation and added missing experiments of Paper C.

Paper D extended Paper C and we provided a rigorous system description of our reduction engine. We presented the developed algorithms for applying adder substitution and circuit verification. Furthermore, we derived upper bounds for the proof length, size, and degree of simple multiplier architectures. In Chap. 6 we further elaborated on proof generation and discussed the effect of different addition orders. We completed the discussion on proof metrics.

In Paper C two different verification techniques for circuit verification were used. Thus two proof certificates in different proof formats were generated. In Paper E, we aimed to merge both proofs, such that one single proof is generated. We derived methods to translate DRUP proofs into PAC, which are complete. PAC proofs cannot fully be translated into DRUP proofs, which leaves gaps in the certification arguments. In Chap. 7, we further elaborated on proof merging. We showed why the DRUP-to-PAC approach of Paper E leads to redundant proof rules and we discussed the need of extension rules. Additionally, we expanded the discussion why the method to convert PAC proofs into DRUP does not lead to complete DRUP proofs.

In Paper F we introduced a new syntax of PAC, which uses indices to name polynomials. Furthermore, we extended PAC of Paper B and added deletion and extension rules. We presented our proof checking tools PACHECK and PASTÈQUE, which are able to check PAC proofs in the new syntax. In Chap. 8 we presented design decisions of our tool PACHECK, in particular we discussed the addition algorithm and the choice of variable sorting. Furthermore, we discussed and modified the experiments of Paper F.

In Chap. 9 we gave a comprehensive evaluation of our developed methods and compared our techniques to the most recent related work. We selected simple and complex multiplier architectures of different input bit-widths and applied verification, certification, and proof checking. Our experiments showed that we are faster than related work but our approach struggles for multipliers that contain certain components.

Tackling these architectures is an interesting future work. Furthermore, we want to make our approach applicable to optimized multipliers, where gate synthesis and technology mapping is applied to reduce the size and computation time of circuits. Additionally we want to investigate floating points and other word-level operators. Another idea for future work is to develop techniques that enhance proof generation and lead to shorter proofs.

Part II

Papers



Paper A

Incremental Column-Wise Verification of Arithmetic Circuits Using Computer Algebra

To be published In the Special Issue on Formal Methods in Computer-Aided Design of the International Journal on Formal Methods in System Design (FMSD) and is currently available as an “Online First Article” ¹.

This article extends and revises work presented in [17, 89, 91], which are published in the Proceedings of the 17th International Conference on Formal Methods in Computer Aided Design (FMCAD 2017) [89], in the Proceedings of the 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2017) [17] and in the Proceedings of the Design, Automation & Test in Europe Conference (DATE 2018) [91].

Authors Daniela Kaufmann, Armin Biere and Manuel Kauers.

Acknowledgement This work is supported by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), Y464-N18, SFB F5004.

Abstract Verifying arithmetic circuits and most prominently multiplier circuits is an important problem which in practice still requires substantial manual effort. The currently most effective approach uses polynomial reasoning over pseudo boolean polynomials. In this approach a word-level specification is reduced by a Gröbner basis which is implied by the gate-level representation of the circuit. This reduction returns zero if and only if the circuit is correct. We give a rigorous formalization of this approach including soundness and completeness arguments. Furthermore we present a novel incremental column-wise technique to verify gate-level multipliers. This approach is further improved by extracting full- and half-adder constraints in the circuit which allows us to rewrite and reduce the Gröbner basis. We also present a new technical theorem which allows us to rewrite local parts of the Gröbner basis. Optimizing the

¹<https://link.springer.com/journal/10703/onlineFirst>

Gröbner basis reduces computation time substantially. In addition we extend these algebraic techniques to verify the equivalence of bit-level multipliers without using a word-level specification. Our experiments show that regular multipliers can be verified efficiently by using off-the-shelf computer algebra tools, while more complex and optimized multipliers require more sophisticated techniques. We discuss in detail our complete verification approach including all optimizations.

A.1 Introduction

Formal verification of arithmetic circuits is important to help to prevent issues like the famous Pentium FDIV bug. Even more than 20 years after detecting this bug the problem of verifying arithmetic circuits and especially multiplier circuits is still considered to be hard. A common approach models the verification problem as a satisfiability (SAT) problem, in which the circuit is translated into a formula in conjunctive normal form (CNF) which is then passed on to SAT-solvers. In order to stimulate the development of fast SAT solving techniques for arithmetic circuit verification, a large set of these benchmarks was generated and the CNF encodings were submitted to the SAT 2016 competition. They are publicly available [12]. The competition results confirmed that mitering of even small multipliers produce very hard SAT problems. The weak performance of SAT solvers on this benchmark set lead to the conjecture that verifying mitering of multipliers and other ring properties after encoding them into CNF needs exponential sized resolution proofs [14], which would imply exponential run-time of CDCL SAT solvers. However, this conjecture was recently rebutted. In [7] it was shown that such ring properties do admit polynomial sized resolution proofs. But since proof search is non-deterministic, this theoretical result still needs to be transferred into practical SAT solving.

Alternative verification techniques use decision diagrams [22, 24], more specifically binary decision diagrams (BDDs) and binary moment diagrams (BMDs) are used for circuit verification. The drawback of BDDs is their high usage of memory for this kind of benchmarks [22]. This issue can be resolved by using BMDs which remain linear in the number of input bits of a multiplier. Actually BMDs and variants of them have been shown to be capable of detecting the Pentium FDIV bug. However, the BMD approach is not robust, it still requires explicit structural knowledge of the multipliers [29]. It is important to determine the order in which BMDs are built, because it has tremendous influence on performance. Actually only a row-wise backward substitution approach seems to be feasible [28], which in addition assumes a simple carry-save-adder (CSA) design.

The currently most effective approach for gate-level verification of arithmetic circuits uses computer algebra [32, 78, 87, 89, 91, 93, 94, 103]. For each gate in the circuit a polynomial is introduced which represents the relation of the gate output and the inputs of the gate. To ensure that variables in the circuit are restricted to boolean values, additional so-called “field polynomials” are introduced. Furthermore the word-level

specification of the multiplier is modeled as a polynomial. If the circuit variables are ordered according to their reverse topological appearance in the circuit, i.e., a gate output variable is greater than the input variables of the gate, then the gate polynomials and field polynomials form a Gröbner basis. As a consequence, the question if a gate-level circuit implements a correct multiplier can be answered by reducing the multiplier specification polynomial by the circuit Gröbner basis. The multiplier is correct if and only if the reduction returns zero.

Related work [32, 103] uses a similar algebraic approach, which is called function extraction. The word-level output of the circuit is rewritten using the gate relations and the goal is to derive a unique polynomial representation of the gate inputs. In order to verify correctness of the circuit this polynomial is then compared to the circuit specification. This rewriting method is essentially the same as Gröbner basis reduction and is able to handle very large clean multipliers but fails on slightly optimized multiplier architectures. The authors of [78, 87, 104] focus on verification of Galois field multipliers using Gröbner basis theory. In contrast we focus in our work [17, 89, 91] on integer multipliers as the authors of [32, 93, 94, 103] do. In [93, 94] the authors propose a sophisticated reduction scheme which is used to rewrite and simplify the Gröbner basis, which as a consequence reduces computation time substantially. Several optimizations are introduced which made their verification technique scale to large multipliers of various architectures [53], but their arguments for soundness and completeness are rather imprecise and neither the tools nor details about experiments are publicly available.

Inspired by these ideas we presented in [89] an incremental column-wise verification technique for integer multipliers where a multiplier circuit is decomposed into columns. In each column the partial products can be uniquely identified and we can define a distinct specification for each slice relating the partial products, incoming carries, slice output and outgoing carries of the slice. We incrementally apply Gröbner basis reduction on the slices to verify the circuit. The incremental column-wise checking algorithm is improved in [17, 91]. The idea in this work is to simplify the Gröbner basis by introducing linear adder specifications. We search for full- and half-adder structures in the gate-level circuit and eliminate the internal gates of the adder structures, with the effect of reducing the number of polynomials in the Gröbner basis. Furthermore we are able to include adder specifications in the Gröbner basis. Reducing by these linear polynomials leads to substantial improvements in terms of computation time.

Alternatively to circuit verification using a word-level specification, it is also common to check the equivalence of a gate-level circuit and a given reference circuit. This technique is extremely important when it is not possible to write down the word-level specification of a circuit in a canonical expression. In [95] equivalence checking of multiplier circuits is achieved by first extracting half-adder circuits from the accumulation of partial products and then checking the equivalence of these extracted half-adder circuits. Proofs of soundness and completeness are lacking. More recently [94] proposes an algebraic variant of combinational equivalence checking based on Gröbner basis theory. It is similar to SAT sweeping [70], and compares the circuits bit-wise, e.g., output bit by output bit, again without soundness nor completeness proof.

As a further contribution we present an extension of our incremental column-wise

verification approach, which can be used to incrementally derive the equivalence of two arbitrary gate-level circuits in a column-wise fashion. We prove soundness and completeness for this method.

This article extends and revises work presented earlier in [17, 89, 91]. Extending [89], we provide a more detailed description of the algebraic approach, including several examples. In Sect. A.4 we introduce additional rewriting methods, called “Partial Product Elimination” and “Adder-Rewriting” [17, 91], which help to further simplify the Gröbner basis. We present the theory behind these rewriting approaches in Sect. A.5 including a theoretical theorem [17], which allows that only a local part of the Gröbner basis is rewritten without losing the Gröbner basis property. In Sect. A.8 we generalize our incremental column-wise verification approach to an incremental equivalence checking approach [91].

For this article we revised our engineering techniques and discuss a new method to derive our column-wise slices in Sect. A.9, which reduces the need of reallocating gates. Furthermore we were able to improve the computation time of the experiments in [89] by adjusting the order of polynomials during printing, cf. Sect. A.9.

A.2 Algebra

Following [17, 32, 78, 87, 89, 91, 93, 94, 103], we model the behavior of a circuit using multivariate polynomials. For each input and output of a logical gate a variable is introduced. The behavior of a gate, i.e., the relation of the gate inputs to the output of a gate is translated into a polynomial. The set of all these polynomials builds a comprehensive description of the circuit. We show that the circuit is correct if and only if the circuit specification, a polynomial describing the relation of the circuit inputs and outputs, is implied by the gate-level polynomials.

The appropriate formalism for such a reasoning is the theory of Gröbner bases [25, 26, 35]. Throughout this section let $\mathbb{K}[X] = \mathbb{K}[x_1, \dots, x_n]$ denote the ring of polynomials in variables x_1, \dots, x_n with coefficients in the field \mathbb{K} .

Definition A.1. A *term* (or *power product*) is a product of the form $x_1^{e_1} \cdots x_n^{e_n}$ for certain non-negative exponents $e_1, \dots, e_n \in \mathbb{N}$. The set of all terms is denoted by $[X]$. A *monomial* is a constant multiple of a term, $\alpha x_1^{e_1} \cdots x_n^{e_n}$ with $\alpha \in \mathbb{K}$. A *polynomial* is a finite sum of monomials.

On the set of terms we fix an order such that for all terms τ, σ_1, σ_2 we have $1 \leq \tau$ and $\sigma_1 \leq \sigma_2 \Rightarrow \tau\sigma_1 \leq \tau\sigma_2$. Such an 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 $\sigma_1 < \sigma_2$ iff there exists an index i with $u_j = v_j$ for all $j < i$, and $u_i < v_i$.

Since every polynomial $p \neq 0$ contains only finitely many terms and they are ordered according to our fixed order $<$, we can determine the largest term in a polynomial. We call it the *leading term* of p and write $\text{lt}(p)$. If $p = c\tau + \cdots$ and $\text{lt}(p) = \tau$, then $\text{lc}(p) = c$ is called the *leading coefficient* and $\text{lm}(p) = c\tau$ is called the *leading monomial* of p . The *tail* of p is defined by $p - c\tau$.

Definition A.2. A nonempty subset $I \subseteq \mathbb{K}[X]$ is called an *ideal* if

$$\forall p, q \in I : p + q \in I \quad \text{and} \quad \forall p \in \mathbb{K}[X] \forall q \in I : pq \in I.$$

If $I \subseteq \mathbb{K}[X]$ is an ideal, then a set $P = \{p_1, \dots, p_m\} \subseteq \mathbb{K}[X]$ is called a *basis* of I if $I = \{q_1 p_1 + \dots + q_m p_m \mid q_1, \dots, q_m \in \mathbb{K}[X]\}$, i.e., if I consists of all the linear combinations of the p_i with polynomial coefficients. We denote this by $I = \langle P \rangle$ and say I is generated by P .

In general, an ideal I has many bases which generate the ideal. We are particularly interested in bases with certain structural properties, called Gröbner bases.

Definition A.3. A basis $G = \{g_1, \dots, g_n\}$ of an ideal $I \subseteq \mathbb{K}[X]$ is called a *Gröbner basis* (w.r.t. the fixed order \leq) if the leading term of every nonzero element of I is a multiple of (at least) one of the leading terms $\text{lt}(g_1), \dots, \text{lt}(g_n)$.

Lemma A.4. Every ideal $I \subseteq \mathbb{K}[X]$ has a Gröbner basis w.r.t. a fixed term order.

Proof. Cor. 6 in Chap. 2 §5 of [35]. □

The following Lemma A.5 describes *Buchberger's Criterion*, which states when a basis of an ideal is a Gröbner basis. Given an arbitrary basis of an ideal, Buchberger's algorithm [25] is able to compute a Gröbner basis for it in finitely many steps. The algorithm is based on repeated computation of so-called S-polynomials.

Lemma A.5. Let $G \subseteq \mathbb{K}[X] \setminus \{0\}$ be a basis of an ideal $I = \langle G \rangle$. We define S-polynomials

$$\text{spol}(p, q) := \text{lcm}(\text{lt}(p), \text{lt}(q)) \left(\frac{p}{\text{lm}(p)} - \frac{q}{\text{lm}(q)} \right)$$

for all $p, q \in \mathbb{K}[X] \setminus \{0\}$, with lcm the least common multiple. Then G is a Gröbner basis of the ideal I if and only if the remainder of the division of $\text{spol}(p, q)$ by G is zero for all pairs $(p, q) \in G \times G$.

Proof. Thm. 6 in Chap. 2 §6 of [35]. □

To reduce the computation effort of Buchberger's algorithm several optimizations exist which decrease the number of S-polynomial computations. We will heavily make use of the following optimization.

Lemma A.6 (Product criterion). If $p, q \in \mathbb{K}[X] \setminus \{0\}$ are such that the leading terms are coprime, i.e., $\text{lcm}(\text{lt}(p), \text{lt}(q)) = \text{lt}(p) \text{lt}(q)$, then $\text{spol}(p, q)$ reduces to zero mod $\{p, q\}$.

Proof. Prop. 4 in Chap. 2 §9 of [35]. □

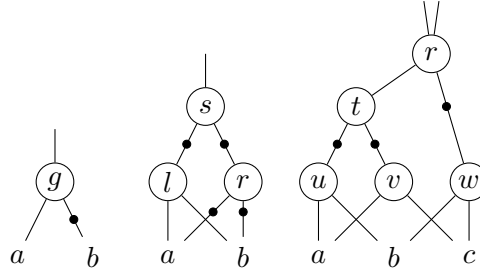


Figure A.1: And-Inverter Graphs (AIGs) [70] used in Ex. A.8 and later in Sect. A.4.

Since $\{p, q\} \subseteq G$, Lemma A.6 suggests that if all leading terms of the polynomials in a basis G of an ideal I are coprime, i.e., we cannot find any pair of polynomials $p, q \in G$ such that $\text{lt}(p)$ and $\text{lt}(q)$ have any variable in common, then the product criterion holds for all pairs of polynomials of G and thus G is automatically a Gröbner basis for the ideal I .

To answer the question if a circuit is correct and hence fulfills its specification we need to check if the specification polynomial is contained in the ideal generated by the circuit relations, as we discuss in detail in Sect. A.3. The theory of Gröbner bases offers a decision procedure for this so-called *ideal membership problem*: Given a polynomial $f \in \mathbb{K}[X]$ and an ideal $I = \langle G \rangle \subseteq \mathbb{K}[X]$, determine if $f \in I$.

Given an arbitrary basis G of the ideal I , it is not so obvious how to check whether the polynomial f belongs to the ideal $I = \langle G \rangle$. However, if G is a Gröbner basis of I , then the membership question can be answered using a multivariate version of polynomial division with remainder, cf. Alg. 1, as derivation procedure. It can be shown that whenever G is a Gröbner basis, then f belongs to the ideal generated by G if and only if the remainder of division of f by G is zero. In the following we will introduce this approach more formally.

Lemma A.7 (Multivariate Division with Remainder). *Let the set of terms be ordered according to a fixed order $<$ and let $P = (p_1, \dots, p_s)$ be an ordered list of polynomials in $\mathbb{K}[X]$. Then every $f \in \mathbb{K}[X]$ can be written as:*

$$f = h_1 p_1 + \dots + h_s p_s + r$$

where $h_1, \dots, h_s, r \in \mathbb{K}[X]$. The remainder r is either zero or is a polynomial $\in \mathbb{K}[X]$, such that no term in r is a multiple of some $\text{lt}(p_i)$. The complete division algorithm is listed in Alg. 1. We call the polynomials h_i the co-factors of f and the polynomial r is called the remainder of f with respect to P .

Proof. Thm. 3 in Chap. 2 §3 of [35]. □

Example A.8. Figure A.1 depicts several And-Inverter-Graphs (AIGs) [70]. A node in an AIG represents logical conjunction of the two inputs, depicted by edges on the lower

Algorithm 1: Multivariate Division Algorithm [35]

Input : p_1, \dots, p_s, f
Output : h_1, \dots, h_s, r

```

1  $h_1 = 0, \dots, h_s = 0, r = 0;$ 
2  $p = f;$ 
3 while  $p \neq 0$  do
4    $i = 1, \text{division} = \text{false};$ 
5   while  $i \leq s \wedge \text{division} = \text{false}$  do
6     if  $\text{lt}(p_i) \mid \text{lt}(p)$  then
7        $h_i = h_i + \text{lt}(p) / \text{lt}(p_i);$ 
8        $p = p - p_i \cdot \text{lt}(p) / \text{lt}(p_i);$ 
9        $\text{division} = \text{true};$ 
10    else
11       $i = i + 1;$ 
12  if  $\text{division} = \text{false}$  then
13     $r = r + \text{lt}(p);$ 
14     $p = p - \text{lt}(p);$ 
15 return  $h_1, \dots, h_s, r$ 

```

half of the node. The output is depicted by an edge in the upper half of the node. An edge containing a marker negates the variable.

Let $\mathbb{K} = \mathbb{Q}$. Hence for the AIG on the left of Fig. A.1, we have the relation $g = a(1-b)$ for all $a, b, g \in \{0, 1\}$. Furthermore, we always have $g(g-1) = a(a-1) = b(b-1) = 0$ since $a, b, g \in \{0, 1\}$. To show that we always have $gb = 0$, it suffices to check if the polynomial $gb \in \mathbb{Q}[g, a, b]$ is contained in the ideal $I \subseteq \mathbb{Q}[g, a, b]$ with

$$I = \langle -g + a(1-b), g(g-1), a(a-1), b(b-1) \rangle.$$

Multivariate polynomial division yields

$$gb = \overset{h_1}{\downarrow} (-b) (-g + a(1-b)) + \overset{h_4}{\downarrow} (-a) b(b-1) + \overset{\text{remainder } r}{\downarrow} 0,$$

with $h_2 = h_3 = 0$, and therefore $gb \in I$ and thus $gb = 0$ in the left AIG of Fig. A.1.

As shown in this example, we can view an ideal $I = \langle G \rangle \subseteq \mathbb{K}[X]$ as an equational theory, where the basis $G = \{g_1, \dots, g_m\}$ defines the set of axioms. The ideal $I = \langle G \rangle$ contains exactly those polynomials f for which the equation “ $f = 0$ ” can be derived from the axioms “ $g_1 = \dots = g_m = 0$ ” through repeated application of the rules $u = 0 \wedge v = 0 \Rightarrow u + v = 0$ and $u = 0 \Rightarrow uw = 0$ (compare to Def. A.2).

Lemma A.9. *If $G = \{g_1, \dots, g_m\}$ is a Gröbner basis, then every $f \in \mathbb{K}[X]$ has a unique remainder r with respect to G . Furthermore it holds that $f - r \in \langle G \rangle$.*

Proof. Prop. 1 in Chap. 2 §6 of [35]. \square

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 A.10. *Let $G = \{g_1, \dots, g_m\} \subseteq \mathbb{K}[X]$ be a Gröbner basis, and let $f \in \mathbb{K}[X]$. Then f is contained in the ideal $I = \langle G \rangle$ iff the remainder of f with respect to G is zero.*

Proof. Cor. 2 in Chap. 2 §6 of [35]. \square

A.3 Ideals associated to Circuits

We consider circuits C with two bit-vectors a_0, \dots, a_{n-1} and b_0, \dots, b_{n-1} of size n as inputs, and a bit-vector s_0, \dots, s_{2n-1} of size $2n$ as output. The circuit is represented by a number of logical gates where the output of some gate may be input to some other gate, but cycles in the circuit are not allowed. Additionally to the variables a_i, b_i, s_i for the inputs and outputs of the circuit, we associate a variable g_1, \dots, g_k to each internal gate output. In our setting let $\mathbb{K} = \mathbb{Q}$. By R we denote the ring $\mathbb{Q}[a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1}, g_1, \dots, g_k, s_0, \dots, s_{2n-1}]$, containing all polynomials in the above variables with coefficients in \mathbb{Q} . At first glance it may seem surprising that we use \mathbb{Q} instead of \mathbb{Z}_2 as ground field although all our variables are restricted to boolean values. The reason for this choice is that we want to verify correctness of integer multiplication. As we will see in Def. A.13, using \mathbb{Q} as base field allows us to describe the desired behavior of the circuit by connecting it to the multiplication in \mathbb{Q} . It would also be possible to use \mathbb{Z}_2 , but in this case, specifying the desired behavior of the circuit in terms of polynomial equations would not be much easier than constructing a circuit in the first place. Such a specification would not be more trustworthy than the circuit that we want to verify.

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

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

The polynomials in R are chosen such that the boolean roots of the polynomials are the solutions of the corresponding gate constraints and vice versa. We denote these polynomials by *gate polynomials*. To ensure that we only find boolean solutions of the polynomials we add the relations $u(u - 1) = 0$ for each variable u . We call this relations *field polynomials*.

Example A.11. The possible boolean solutions for the gate constraint $p_{00} = a_0 \wedge b_0$ of Fig. A.2 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.

Since the logical gates in a circuit are functional, the values of all the variables $g_1, \dots, g_k, s_0, \dots, s_{2n-1}$ in a circuit are uniquely 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.

Definition A.12. 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 can easily be verified 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. Therefore, the circuit fulfills a certain specification if and only if the polynomial relation corresponding to the specification of the circuit is contained in the ideal $I(C)$.

Definition A.13. A circuit C is called a *multiplier* if the word-level specification

$$\sum_{i=0}^{2n-1} 2^i s_i - \left(\sum_{i=0}^{n-1} 2^i a_i \right) \left(\sum_{i=0}^{n-1} 2^i b_i \right) \in I(C).$$

Thus checking whether a given circuit C is a correct multiplier reduces to an ideal membership test. Definition A.12 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 A.14. Let C be a circuit. Let $G \subseteq R$ be the set which contains for each gate of C the corresponding polynomial of Eqn. (A.1), where the variable u is replaced by the output variable and v, w are replaced by the input variables of the gate. Furthermore G contains the polynomials $a_i(a_i - 1)$ and $b_i(b_i - 1)$ for $0 \leq i < n$, called *input field polynomials*. Then the ideal $\langle G \rangle \subset R$ is denoted by $J(C)$.

Hence G is a basis for the ideal $J(C)$ and we can decide membership using Gröbner bases theory. Assume that we have a verifier which checks for a given circuit C and a given specification polynomial $p \in R$ if p is contained in the ideal $J(C)$. Because it holds that $J(C) \subseteq I(C)$, such a verifier is sound. To show that the verifier is also complete, we further need to show $J(C) \supseteq I(C)$. For doing so, we recall an important observation shown for instance in [78, 100].

Theorem A.15. Let C be a circuit, and let G be as in Def. A.14. 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 is a Gröbner basis with respect to the ordering \leq .

Proof. By the restrictions on the term order and the form of Eqns. (A.1), the leading term of each gate polynomial is simply the output variable of the corresponding gate. Furthermore, the leading terms of the input field polynomials $a_i(a_i - 1)$ and $b_i(b_i - 1)$ are a_i^2 and b_i^2 . Hence all leading terms are coprime and therefore, by Lemma A.6, division of $\text{spol}(p, q)$ by $\{p, q\}$ has remainder zero for any choice $p, q \in G$. Since $\{p, q\} \subseteq G$ for all $p, q \in G$, division of $\text{spol}(p, q)$ by G gives the remainder zero for all $p, q \in G$, and then, by Lemma A.5, the claim follows. \square

Theorem A.16. *For all acyclic circuits C , we have $J(C) = I(C)$.*

Proof. “ \subseteq ” (soundness): Immediately follows from the definition of $J(C)$.

“ \supseteq ” (completeness): Let $p \in R$ be a polynomial with $p \in I(C)$. We show that $p \in J(C)$. Since C is acyclic, we can order the variables according to the needs of Thm. A.15. Hence by Thm. A.15 we can derive a Gröbner basis G for $J(C)$. Let r be the remainder of division of p by G . Thus $p - r \in J(C)$ by Lemma A.9, and $r \in J(C) \iff p \in J(C)$. Then, since $J(C) \subseteq I(C)$ it holds that $p - r \in I(C)$. By $p \in I(C)$ and $p - r \in I(C)$ it follows that $r \in I(C)$. Thus we need to show $r \in J(C)$.

By the choice of the ordering of the terms and the observations about the leading terms in G made in the proof of Thm. A.15, from Lemma A.9 it also follows that r only contains input variables $a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1}$, and each of them has a maximum degree of one. Simultaneously, $r \in I(C)$ implies that all evaluations of r for all choices $a_i, b_j \in \{0, 1\}$ are zero.

We show $r = 0$, and thus $r \in J(C)$. Assume $r \neq 0$. Suppose m is a monomial of r with a minimal number of variables, including the case that m is a constant. Since the exponents are at most one, no two monomials in r contain exactly the same variables. Now select a_i (b_j) to evaluate to 1 iff $a_i \in m$ ($b_j \in m$). Hence all monomials of r except m evaluate to zero and thus vanish. By this choice r evaluates to the (non-zero) coefficient of m , contradicting $r \in I(C)$. Thus $r = 0$. \square

Example A.17. In contrast to our definition of a circuit, where both input bit-vectors have the same length, Fig. A.2 shows a 3×2 -bit multiplier. The leading terms of the polynomials in the right column, read from top to bottom, follow a reverse topological lexicographic ordering. Hence these polynomials form a Gröbner basis.

We conclude this section with the following simple but important observations. First, the ideal $I(C)$ is a so-called vanishing ideal. Therefore, it follows that $J(C)$ is a radical ideal. Hence testing ideal membership of the specification is sufficient for verifying the correctness of a circuit, and we do not need to apply the stronger radical membership test (cf. Chap. 4 §2 of [35]).

Second, since it holds that $I(C) = J(C)$ contains all the *field polynomials* $u(u - 1)$ for all variables u , not only for the inputs, we may add them to G .

Third, in the Gröbner basis G for gate-level circuits defined as given in Def. A.14 using Eqn. (A.1) it holds that all polynomials have leading coefficient ± 1 . Thus during reduction (division) no coefficient outside of \mathbb{Z} (with non-trivial denominator) is introduced. Hence all coefficient computations actually remain in \mathbb{Z} . This formally shows

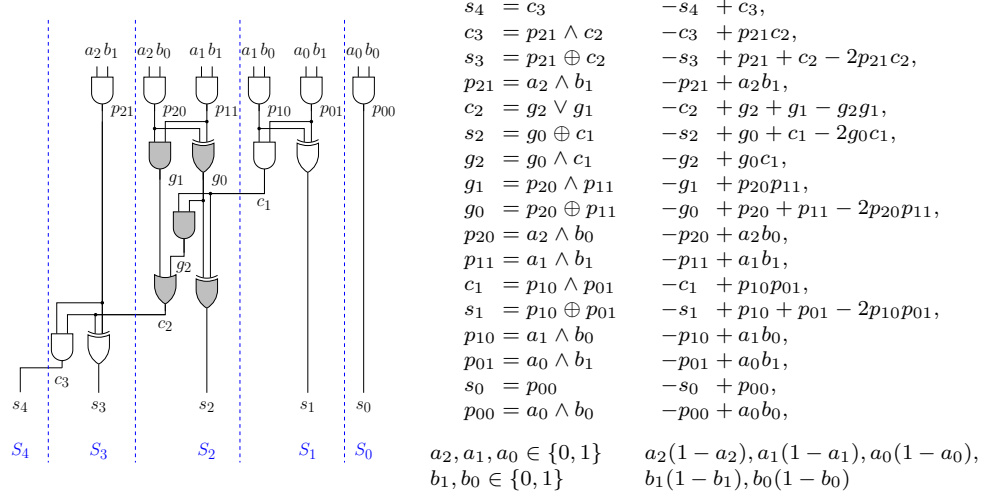


Figure A.2: A 3x2-bit gate-level multiplier circuit, gate constraints, and polynomials. Colored gates represent a full adder, cf. Sect. A.5. Dashed lines depict column-wise slicing, cf. Sect. A.7.

that the implementations, e.g., those from [93, 103], used for proving ideal membership to verify properties of gate-level circuits, actually can rely on computation in \mathbb{Z} without loosing soundness nor completeness. Of course it needs to hold that the same term order as in Thm. A.15 is used.

Fourth, we do not need \mathbb{Z} as coefficient ring if we use computer algebra systems, we can simply choose any field containing \mathbb{Z} , e.g., \mathbb{Q} , which actually improves computation, because \mathbb{Z} is not a field and ideal theory over rings is harder than ideal theory over fields. In our experiments, using rational coefficients made a huge difference for Singular [38] (but did not show any effect in Mathematica [102]).

Fifth, because the leading terms of G contain only one variable, computing a remainder with respect to G has the same effect as substituting each leading term with the corresponding tail until no further substitution is possible.

Sixth, given a circuit C , checking whether an assignment of the inputs exists, which yields a certain value at an output is actually the same as (circuit) SAT, and hence is NP complete:

Corollary A.18. *Consider the problem to decide, for a given polynomial $p \in \mathbb{Q}[X]$ and a given Gröbner basis $G \subseteq \mathbb{Q}[X]$, whether $p \in \langle G \rangle$. Taking the bit-size of p and G in the natural encoding as a measure for the problem size, this problem is co-NP-hard.*

Proof. Circuit SAT is the problem to decide for a given circuit with n gates and one output bit whether it produces the output 1 for at least one choice of inputs. This problem is known to be NP-hard. Consequently, the problem of deciding whether a given circuit with n gates and one output bit s produces the output 1 for every choice of inputs is co-NP-hard. A circuit C returns 1 for every input iff $s - 1 \in J(C)$. As the Gröbner basis G for the circuit C has essentially the same size as C , the circuit problem can

be solved with at most polynomial overhead if we have an algorithm for solving the membership problem. \square

The main point of this corollary is not that ideal membership is difficult, but that it remains difficult even if we assume to be given a Gröbner basis of the ideal as part of the input. For other results on the complexity of the ideal membership problem, see [2, 57].

As a final remark, in the case when a polynomial g is not contained in an ideal $I = \langle G \rangle$, i.e., the remainder of dividing g by G is not zero, the last part in the proof of Thm. A.16, where the “smallest” monomial m is evaluated, allows us 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.

We claim that this section shows the first formalization of not only soundness but also completeness arguments for recent successful algebraic approaches [93, 103]. In previous work soundness and completeness was formally shown too but only for other polynomial rings, i.e., over \mathbb{F}_{2^q} to model circuits which implement Galois-field multipliers [78, 87], or for polynomial rings over \mathbb{Z}_{2^q} which model arithmetic circuit verification using overflow semantics [100]. In the work of [103] soundness and completeness is discussed too, but refers to [78, 87] instead of showing proofs, which as discussed above uses coefficients in \mathbb{F}_{2^q} and not \mathbb{Z} , the coefficient ring the approach [103] is actually working with.

A.4 Optimizations

In this section we extend the “XOR-Rewriting”, “Common-Rewriting” and “Vanishing Constraints” optimizations of [89] by the additional rewriting techniques of “Adder-Rewriting” and “Partial Product Elimination” [17, 91]. Picking up the statement of Cor. A.18, simply reducing the specification polynomial in the constructed Gröbner basis of the circuit generally leads to an exponential number of monomials in the intermediate reduction results. This conjecture was also made in [93]. Thus in practice to efficiently use polynomial reduction for verification of specific circuits tailored heuristics which rewrite Gröbner bases and hence improve the reduction process become very important to speed up computation. The (non-reduced) Gröbner basis of an ideal is not unique, thus some Gröbner bases may be better than others, for instance much smaller. A natural choice among all the Gröbner bases is the unique reduced Gröbner basis [35], but it was shown empirically in [91] that the computation of this basis for multipliers is not feasible in practice, e.g., the computation of the unique reduced Gröbner basis for a 4-bit multiplier took more than 20 minutes.

In [93] a logic reduction rewriting scheme consisting of *XOR-Rewriting* and *Common-Rewriting* is proposed which helps to reduce the number of monomials by partially reducing the Gröbner basis. Furthermore several specific monomials are eliminated which fulfill a certain *Vanishing Constraint*.

The technique *XOR-Rewriting* of [93] eliminates all variables of the Gröbner basis which are neither an input nor an output of an XOR-gate. Also the primary input and output variables of the circuit are not eliminated in the Gröbner basis.

In our setting circuits are usually given as AIGs, hence we adopt this rewriting for AIGs by matching XOR (XNOR) patterns in the AIG which represent an XOR (XNOR) gate. This means we want to find a set of nodes for which the relation $s = \overline{(a \wedge b)} \wedge (\bar{a} \wedge \bar{b})$ holds. We eliminate internal variables of these structures and define the polynomial of the XOR (XNOR) output directly in terms of the grandchildren.

Example A.19. The middle AIG in Fig. A.1 depicts an XOR constraint. For this structure we only use the polynomial $-s+a+b-2ab$ for describing the logical constraint instead of the polynomials $-l+ab$, $-r+(1-a)(1-b)$, and $-s+(1-l)(1-r)$. This deletes polynomials containing the variables l, r from the Gröbner basis, unless they are used as an input of further gates.

After applying XOR-Rewriting the *Common-Rewriting* [93] technique further simplifies the Gröbner basis by eliminating all variables which are used exactly once as an input of a further gate. This technique can be compared to bounded variable elimination in SAT [39] after encoding a circuit to a CNF using, e.g., Tseitin encoding. This approach would also eliminate all variables in the CNF representing gates in the circuit having only one parent [56].

Example A.20. The right AIG of Fig. A.1 contains several variables occurring only once, hence Common-Rewriting eliminates gates t, u, v , and w . Thus the relation of r is directly expressed in terms of a, b, c .

Although the concepts of XOR-Rewriting and Common-Rewriting seem rather intuitive in the sense that we can simply rewrite and delete polynomials from the Gröbner basis, we need sophisticated algebraic reasoning, i.e., elimination theory of Gröbner bases. We will introduce this theory in Sect. A.5, but before doing so we want to complete the discussion of possible optimizations.

A further optimization presented in [93] was to add vanishing constraints, i.e., polynomials which are PCCs of the circuit C and because they are contained in $I(C)$, they can be added to the Gröbner basis. In [93] a specific constraint was called the *XOR-AND Vanishing Rule*, denoting that an XOR-gate and AND-gate which have the same input can never be 1 at the same time. An XOR- and AND-gate with the same inputs logically represent a half-adder, where the XOR-gate represents the sum output and the AND-gate represents the carry output. Because a half-adder only sums up two bits, it can never happen that the sum output and carry output is 1 at the same time.

Example A.21. In the middle AIG of Fig. A.1 the variable l represents an AND-gate and s represents an XOR-gate. Both have a, b as input. Hence we can deduce $sl = 0$.

We adapt this rule by searching for (negative) children or grandchildren of specific AND-gates in the circuit. We add a corresponding polynomial to our Gröbner basis which deletes redundant monomials in intermediate reduction results.

Additionally to the above optimizations which we more or less adopted of [93], we presented in [17, 91] a further optimization called *Adder-Rewriting*, which is also based on elimination theory of Gröbner basis. The core idea is to simplify the Gröbner basis by introducing linear adder specifications.

Definition A.22. A sub-circuit C_S of a circuit C is a *full-adder* if

$$-2c - s + a + b + i \text{ is a PCC for } C$$

for outputs c, s and inputs a, b, i of C_S and a *half-adder* if

$$-2c - s + a + b \text{ is a PCC for } C.$$

We search for such sub-circuits representing full- and half-adders in the gate-level circuit C . Then we eliminate the internal variables of these sub-circuits, cf. Sect. A.5, which has the effect that the linear adder specifications are included in the Gröbner basis. Reducing by these linear polynomials leads to substantial improvements in terms of computation time. Furthermore we will also add a polynomial representing the relation of s to the inputs a, b, i , because there are no restrictions on s . It can be used multiple times as a child of a gate and hence we need a relation for it. In general, assuming that the carry output c is always larger than the sum output s , the intermediate reduction polynomials includes the term $2c + s$ before we reduce c . Using the adder specification s is canceled in parallel during the reduction of c . Hence in certain multiplier architectures which consist only of full- and half-adders we never have to reduce s , cf. Sect. A.10. But we have to include polynomials with leading term s , otherwise we lose completeness of our approach.

In [106] a similar strategy is given which detects embedded MAJ3 and XOR3 gates. In this approach the Gröbner basis of the circuit is not simplified, but the MAJ3 and XOR3 gates are used to receive a more efficient reduction order.

Example A.23. The middle AIG in Fig. A.1 shows a half adder with outputs l and s as carry and sum and inputs a, b . Hence we can derive the relations $-2l - s + a + b$ and $-s + a + b - 2ab$. In Fig. A.2 the filled gates describe a full-adder. In this case we can obtain the specification $-2c_2 - s_2 + p_{20} + p_{11} + c_1$ by elimination of g_0, g_1, g_2 .

We apply the optimizations in the following order: Adder-Rewriting, XOR-Rewriting, Common-Rewriting, Adding Vanishing Constraints. We start by eliminating variables from bigger parts of the circuit and continue with rewriting smaller parts and only in the end we add polynomials to the Gröbner basis.

In [91] we introduced a rewriting method which is different from the optimizations above, because in *Partial Product Elimination* we change the circuit specification. In multipliers where a partial product is simply the conjunction of two input bits, we find exactly n^2 polynomials, representing the corresponding AND-gates.

We can eliminate these polynomials by cutting off these gates from the circuit and verify them separately, e.g., we search for them in the AIG, but do not introduce separate polynomials $p_{i,j} = a_i b_j$. Hence we change the specification of the multiplier from Def. A.13 to the specification given in Cor. A.24.

Corollary A.24. *A circuit C is a multiplier if*

$$\sum_{i=0}^{2n-1} 2^i s_i - \sum_{i,j=0}^{n-1} 2^{i+j} p_{i,j} \in I(C) \quad \text{with } p_{i,j} = a_i b_j.$$

We can easily check that the specifications of Cor. A.24 and Def. A.13 are equivalent, when we expand the sums and replace every occurring of $p_{i,j}$ with $a_i b_j$ in Cor. A.24.

This approach works only in multipliers with a simple partial product generation, in multipliers using, e.g., Booth encoding [85] these patterns do not exist, but it might be possible to find similar patterns in this situation too.

In the following we show how rewriting techniques, which are based on variable elimination can be applied to circuit verification.

A.5 Variable Elimination

Section A.4 actually relies on elimination theory of Gröbner bases to justify our rewriting techniques. This section provides more details about this theory and also presents a theorem which allows to rewrite only local parts of the Gröbner basis following [17]. To apply these rewriting techniques the circuit is split into two parts by extracting a sub-circuit, which is then rewritten, without changing the rest of the circuit. For example Adder-Rewriting is applied on an extracted full- or half-adder and XOR-Rewriting is used for nodes in the AIG describing an XOR-constraint. Consequently also the overall ideal $I(C)$ and the Gröbner basis G are split into two parts. In the extracted sub-circuit we want to eliminate redundant internal variables, i.e., variables occurring only inside the sub-circuit. For this purpose we use the elimination theory of Gröbner bases [35].

Recall, that if $I \subseteq \mathbb{Q}[X]$ and $J \subseteq \mathbb{Q}[X]$ are ideals, then their sum is the set $I + J = \{f + g \mid f \in I, g \in J\}$, which in fact is also an ideal in $\mathbb{Q}[X]$.

Lemma A.25. *Let $I = \langle f_1, \dots, f_r \rangle$ and $J = \langle g_1, \dots, g_s \rangle$ be two ideals in $\mathbb{Q}[X]$. Then $I + J = \langle f_1, \dots, f_r, g_1, \dots, g_s \rangle$. In particular $\langle f_1, \dots, f_r \rangle = \langle f_1 \rangle + \dots + \langle f_r \rangle$.*

Proof. Prop. 2 and Cor. 3 in Chap. 4 §3 of [35]. □

In the simple case where all occurring polynomials are linear, the effect of elimination theory can be easily illustrated with Gaussian elimination.

Example A.26 (Gaussian elimination). Let us consider the following system of three linear equations in $\mathbb{Q}[x, y, z]$:

$$\begin{aligned} 2x + 4y - 3z + 4 &= 0 \\ 3x + 7y - 3z + 2 &= 0 \\ 2x + 5y - 4z + 5 &= 0 \end{aligned}$$

Let V be the vector space consisting of all \mathbb{Q} -linear combinations of the polynomials on the left-hand side, then each possible root $(x, y, z) \in \mathbb{Q}^3$ of the above system is also a

root of each polynomial contained in V . In this sense, V contains all linear polynomials whose solutions can be deduced from the roots of the system, i.e., the polynomials generating V .

If we are only interested in polynomials of V in which the variable x does not occur, we can triangularize the above system using Gaussian elimination. This for example leads to the equivalent system:

$$\begin{aligned}x + 2y - 2z + 3 &= 0 \\y + 3z - 7 &= 0 \\z - 2 &= 0\end{aligned}$$

In Gaussian elimination new polynomials are derived by applying linear combinations of the original polynomials. Hence the polynomials on the left-hand side belong to the vector space V . We see that two polynomials do not contain x . In fact, every element of V which does not contain x can be written as a linear combination of the polynomials $y + 3z + 2$ and $z + 1$ which are free of x .

Since Gaussian elimination is defined only for linear equations, we cannot use it for our setting, but using Gröbner bases theory we can extend the reasoning in the example above to systems of nonlinear equations.

In linear polynomials a term consists of a single variable, hence for triangularization we only have to order the terms in such a way that the variables which we want to eliminate are the largest terms. This ordering is generalized to multivariate terms by introducing an elimination order on the set of terms. In the following assume that we want to eliminate the variables belonging to a subset Z of X .

Definition A.27. [35] Let $X = Y \dot{\cup} Z$. An order $<$ on the set of terms of $[X]$ is called *elimination order* for Z if it holds for all terms σ, τ where a variable from Z is contained in σ but not in τ , we obtain $\tau < \sigma$. We denote this ordering by $Y < Z$.

In the case that $Z = \{x_1, \dots, x_i\}$ and $Y = \{x_{i+1}, \dots, x_n\}$, the lexicographic term order is such an elimination order. In Ex. A.26 the elimination order $Y < Z$ is defined by a lexicographic ordering with $Y = \{y, z\}$ and $Z = \{x\}$.

Definition A.28. [35] Assume an ideal $I \subseteq \mathbb{Q}[X] = \mathbb{Q}[Y, Z]$. The ideal where the Z -variables are eliminated is the *elimination ideal* $J \subseteq \mathbb{Q}[Y]$ defined by

$$J = I \cap \mathbb{Q}[Y].$$

Theorem A.29. [35] Given an ideal $I \subseteq \mathbb{Q}[X] = \mathbb{Q}[Y, Z]$. Further let G be a Gröbner basis of I with respect to an elimination order $Y < Z$. Then the set

$$H = G \cap \mathbb{Q}[Y]$$

is a Gröbner basis of the elimination ideal $J = I \cap \mathbb{Q}[Y]$, in particular $\langle H \rangle = J$.

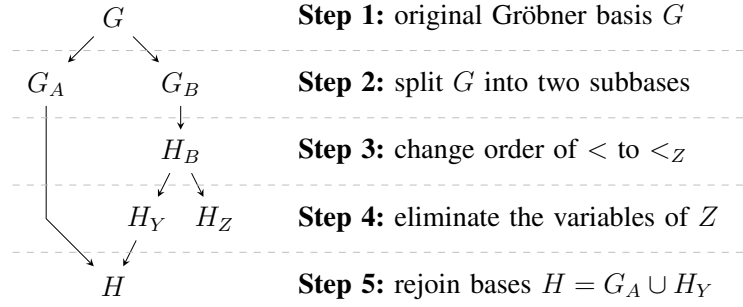


Figure A.3: Each step of the elimination procedure of the proof of Thm. A.30.

The requirements of Thm. A.29 demand that we need to calculate a new Gröbner basis H w.r.t. to an elimination order $Y < Z$ for our circuit C . In general this means that we really need to apply Buchberger's algorithm and cannot simply rely on the product criterion anymore as we did for G . Since Buchberger's algorithm is computationally expensive [35], this is practically infeasible. In [17, 91] we derived a method which allows that we split G into two smaller Gröbner basis G_A and G_B , where $\langle G_B \rangle$ defines the ideal generated by the gate polynomials of the extracted sub-circuit. The following theorem shows that in order to compute a basis of the elimination ideal $J = \langle G \rangle \cap \mathbb{Q}[Y]$ it suffices to compute a basis of the elimination ideal $\langle G_B \rangle \cap \mathbb{Q}[Y]$.

Theorem A.30. *Let $G \subseteq \mathbb{Q}[X] = \mathbb{Q}[Y, Z]$ be a Gröbner basis with respect to some term order $<$. Let $G_A = G \cap \mathbb{Q}[Y]$ and $G_B = G \setminus G_A$. Let $<_Z$ be an elimination order for Z which agrees with $<$ for all terms that are free of Z , i.e., terms free of Z are equally ordered in $<$ and $<_Z$. Suppose that $\langle G_B \rangle$ has a Gröbner basis H_B with respect to $<_Z$ which is such that every leading term in H_B is free of Z or free of Y . Let $H_B = H_Y \cup H_Z$, such that H_Z consists of all polynomials with leading terms in Z and $H_Y = H_B \setminus H_Z$ contains the remaining polynomials with leading terms in Y . Then*

1. $\langle G \rangle \cap \mathbb{Q}[Y] = (\langle G_A \rangle + \langle G_B \rangle) \cap \mathbb{Q}[Y] = \langle G_A \rangle + (\langle G_B \rangle \cap \mathbb{Q}[Y])$.
2. $H = G_A \cup H_Y$ is a Gröbner basis for $\langle G_A \rangle + (\langle G_B \rangle \cap \mathbb{Q}[Y])$ w.r.t. the ordering $<_Z$.

Proof. 1) The steps of the elimination process of this proof are depicted in Fig. A.3. Since $Y <_Z Z$, it follows that the polynomials in H_Y cannot contain any variable of Z . Furthermore by definition G_A does not contain any polynomial containing Z -variables, hence variables of Z only occur in H_Z .

By Lemma A.25 we derive

$$\begin{aligned} \langle G \rangle &= \langle G_A \rangle + \langle G_B \rangle = \langle G_A \rangle + \langle H_B \rangle \\ &= \langle G_A \rangle + \langle H_Y \rangle + \langle H_Z \rangle = \langle G_A \cup H_Y \rangle + \langle H_Z \rangle. \end{aligned}$$

By $GB(S, o)$ we denote an arbitrary Gröbner basis for S w.r.t. an ordering o . Changing

an arbitrary basis into a Gröbner basis does not affect the ideal, hence

$$\begin{aligned}\langle G_A \cup H_Y \rangle + \langle H_Z \rangle &= \langle GB(G_A \cup H_Y, <_Z) \rangle + \langle H_Z \rangle \\ &= \langle GB(G_A \cup H_Y, <_Z) \cup H_Z \rangle.\end{aligned}$$

Furthermore $GB(G_A \cup H_Y, <_Z) \cup H_Z$ is a Gröbner basis, because all S-polynomials of pairs of polynomials p, q reduce to zero:

1. $p, q \in GB(G_A \cup H_Y, <_Z)$: By Lemma A.5, $\text{spol}(p, q)$ reduces to zero.
2. $p \in GB(G_A \cup H_Y, <_Z), q \in H_Z$: The leading terms of H_Z contain only variables of Z , whereas the polynomials $G_A \cup H_Y$ do not contain any variable of Z . Hence by Lemma A.6, $\text{spol}(p, q)$ reduces to zero.
3. $p, q \in H_Z$: Since $H_B = H_Y \cup H_Z$ is a Gröbner basis, it holds that $\text{spol}(p, q)$ reduces to zero w.r.t. H_B . Consequently it reduces to zero w.r.t. $G_A \cup H_B = G_A \cup H_Y \cup H_Z$. Since each leading term of $G_A \cup H_Y$ is a multiple of a leading term in $GB(G_A \cup H_Y, <_Z)$, $\text{spol}(p, q)$ reduces to zero w.r.t. $GB(G_A \cup H_Y, <_Z) \cup H_Z$.

Combining the above results we conclude that $GB(G_A \cup H_Y, <_Z) \cup H_Z$ is a Gröbner basis for the ideal $\langle GB(G_A \cup H_Y, <_Z) \cup H_Z \rangle = \langle G \rangle$. Following Thm. A.29 we receive

$$\begin{aligned}(\langle G_A \rangle + \langle G_B \rangle) \cap \mathbb{Q}[Y] \\ &= \langle GB(G_A \cup H_Y, <_Z) \cup H_Z \rangle \cap \mathbb{Q}[Y] \\ &= \langle GB(G_A \cup H_Y, <_Z) \rangle.\end{aligned}$$

Since computation of a Gröbner basis does not change the ideal, we have

$$\langle GB(G_A \cup H_Y, <_Z) \rangle = \langle G_A \cup H_Y \rangle = \langle G_A \rangle + \langle H_Y \rangle.$$

Because the set H_Y does not contain any variable of Z , it follows

$$\langle H_Y \rangle = \langle H_B \rangle \cap \mathbb{Q}[Y] = \langle G_B \rangle \cap \mathbb{Q}[Y]$$

Altogether by composing the derived results we obtain

$$(\langle G_A \rangle + \langle G_B \rangle) \cap \mathbb{Q}[Y] = \langle G_A \rangle + (\langle G_B \rangle \cap \mathbb{Q}[Y]).$$

2) We need to prove that for every term $\tau \in [Y]$ which is also a leading term of a polynomial in $\langle G \rangle$ it follows that there exists a polynomial $f \in G_A \cup H_Y$ such that $\text{lt}(f) \mid \tau$. Let τ be such a term.

Because G is a Gröbner basis it holds that there exists a $g \in G$ with $\text{lt}(g) \mid \tau$. Since $G = G_A \cup G_B$ it consequently follows that either $g \in G_A$ or $g \in G_B$:

1. $g \in G_A$: It immediately follows that $g \in G_A \cup H_Y$, hence $f := g$.

2. $g \in G_B$: Since $\langle G_B \rangle = \langle H_B \rangle$, there exists an element $h \in H_B$ with $\text{lt}(h) \mid \text{lt}(g)$. From this it follows that $\text{lt}(h) \mid \tau$. Since $\tau \in [Y]$ it further holds that $\text{lt}(h) \in [Y]$. Hence $h \in H_Y$ and altogether $h \in G_A \cup H_Y$. In this case $f := h$.

So for each $g \in G$ we find $f \in G_A \cup H_Y$ whose leading term divides τ . \square

The above theorem allows that we simply add the Gröbner basis H_Y of the elimination ideal of the extracted sub-circuit $\langle H_Y \rangle = \langle H_B \rangle \cap \mathbb{Q}[Y] = \langle G_B \rangle \cap \mathbb{Q}[Y]$ to the Gröbner basis G_A of the remaining circuit and obtain again a Gröbner basis, preventing that we have to compute a new Gröbner basis w.r.t. to an elimination order for the whole circuit C . Actually we only have to really compute one “small” local Gröbner basis H_B . Although in principle we can choose Z arbitrarily, we apply the idea to sets of variables that only occur locally in the circuit. One source of such variables are intermediate results of adders.

Example A.31. We want to apply Adder-Rewriting on the full adder in the circuit C of Fig. A.2, which is depicted by the colored gates. The full adder has outputs c_2 (carry) and s_2 (sum) and three inputs p_{20}, p_{11}, c_1 . The internal gates g_2, g_1, g_0 are not used outside the full adder structure, hence we want to eliminate them and include the specification $2c_2 + s_2 - p_{20} - p_{11} - c_1$ in the Gröbner basis G .

The Gröbner basis G which is depicted by polynomials in the right column in Fig. A.2 is split such that $G_A = G \setminus G_B$ and

$$G_B = \{-g_0 + p_{20} + p_{11} - 2p_{20}p_{11}, \quad -g_1 + p_{20}p_{11}, \quad -g_2 + c_1g_0, \\ -s_2 + c_1 + g_0 - 2c_1g_0, \quad -c_2 + g_1 + g_2 - g_1g_2\}$$

We apply variable elimination only in G_B . For this let $Z = \{g_2, g_1, g_0\}$. According to the requirements of Thm. A.29 we need to find an elimination order $<_Z$ such that $Y < Z$. So far we used in Ex. A.17 a lexicographic term ordering $<$ with

$$b_0 < b_1 < a_0 < a_1 < a_2 < p_{00} < s_0 < p_{01} < p_{10} < s_1 < c_1 < \\ p_{11} < p_{20} < g_0 < g_1 < g_2 < s_2 < c_2 < p_{21} < s_3 < c_3 < s_4$$

We choose $<_Z$ such that $<$ and $<_Z$ restricted on Y are equal, i.e., we move g_0, g_1, g_2 to be the largest variables in the lexicographic ordering, but do not change the order of the remaining variables.

We compute a Gröbner basis H_B w.r.t. $<_Z$. During the computation we use the notation $f \xrightarrow{P} r$, meaning that r is the remainder f with respect to P . For simplification we immediately reduce higher powers without showing this reduction by the field polynomials explicitly. Initially H_B contains:

$$\begin{aligned} f_1 &:= -g_0 - 2p_{20}p_{11} + p_{20} + p_{11}, & f_2 &:= -g_1 + p_{20}p_{11}, \\ f_3 &:= -g_2 + g_0c_1, & f_4 &:= -2c_1g_0 + g_0 - s_2 + c_1, \\ f_5 &:= -g_2g_1 + g_2 + g_1 - c_2 \end{aligned}$$

According to Buchberger’s algorithm [25] we consider all possible pairs $(f_i, f_j) \in H_B \times H_B$ and compute $\text{spol}(f_i, f_j) \xrightarrow{H_B} r$. If r is not zero, we add r to H_B . This step is repeated until all $\text{spol}(f_i, f_j)$ for $(f_i, f_j) \in H_B \times H_B$ reduce to zero.

Initially we only have to explicitly compute remainders of $\text{spol}(f_1, f_4)$, $\text{spol}(f_2, f_5)$ and $\text{spol}(f_3, f_5)$, because all other S-Polynomials reduce to zero according to the product criterion, cf. Lemma A.6.

$$\begin{aligned} \text{spol}(f_1, f_4) &= 2c_1f_1 - f_4 = -g_0 + s_2 - 4p_{20}p_{11}c_1 + 2p_{20}c_1 + 2p_{11}c_1 - c_1 \\ &\xrightarrow{\{f_1\}} s_2 - 4p_{20}p_{11}c_1 + 2p_{20}p_{11} + 2p_{20}c_1 - p_{20} + 2p_{11}c_1 - p_{11} - c_1 =: f_6 \end{aligned}$$

The non-zero remainder f_6 of $\text{spol}(f_1, f_4)$ is added to H_B . Since $\text{lt}(f_6)$ is coprime to all other leading terms of H_B , all $\text{spol}(f_i, f_6)$ reduce to zero, cf. Lemma A.6.

$$\begin{aligned} \text{spol}(f_2, f_5) &= g_2f_2 - f_5 = g_2p_{20}p_{11} - g_2 - g_1 + c_2 \\ &\xrightarrow{\{f_3\}} -g_1 + g_0p_{20}p_{11}c_1 - g_0c_1 + c_2 \\ &\xrightarrow{\{f_2\}} g_0p_{20}p_{11}c_1 - g_0c_1 + c_2 - p_{20}p_{11} \\ &\xrightarrow{\{f_1\}} c_2 + 2p_{20}p_{11}c_1 - p_{11}c_1 + p_{20}c_1 - p_{20}p_{11} =: f_7 \end{aligned}$$

We add f_7 to H_B and we again apply the product criterion for all S-Polynomials containing f_7 .

$$\begin{aligned} \text{spol}(f_3, f_5) &= g_1f_3 - f_5 = -g_2 + g_1g_0c_1 - g_1 + c_2 \\ &\xrightarrow{\{f_3\}} g_1g_0c_1 - g_1 - g_0c_1 + c_2 \\ &\xrightarrow{\{f_2\}} g_0p_{20}p_{11}c_1 - g_0c_1 + c_2 - p_{20}p_{11} \\ &\xrightarrow{\{f_1\}} c_2 + 2p_{20}p_{11}c_1 - p_{11}c_1 + p_{20}c_1 - p_{20}p_{11} \xrightarrow{\{f_7\}} 0 \end{aligned}$$

At this point the algorithm terminates, because now all S-Polynomials reduce to zero. Thus $H_B = \{f_1, f_2, f_3, f_4, f_5, f_6, f_7\}$ is a Gröbner basis for $\langle H_B \rangle$.

Although H_B is already a Gröbner basis, we will modify it to cover our needs. It is always allowed to add polynomials of $\langle H_B \rangle$ to H_B without violating the Gröbner basis property. In order to add the specification of the full adder to H_B we construct $f_8 := 2f_7 + f_6 = 2c_2 + s_2 - p_{20} - p_{11} - c_1$ and add it to H_B .

To reduce the size of the Gröbner basis H_B we eliminate unnecessary polynomials. Lemma 3 in Chap. 2 §7 of [35] tells us that we can remove a polynomial p from our Gröbner Basis H_B whenever we have a further polynomial $q \in H_B$ such that $\text{lt}(q) \mid \text{lt}(p)$. Thus we can eliminate f_4, f_5 and f_7 and our final Gröbner basis H_B w.r.t. $<_Z$ is

$$\begin{aligned} H_B = \{ & \textcolor{blue}{g_0 + 2p_{20}p_{11} - p_{20} - p_{11}}, \quad \textcolor{blue}{g_1 - p_{20}p_{11}}, \quad \textcolor{blue}{g_2 + g_0c_1}, \\ & s_2 - 4p_{20}p_{11}c_1 + 2p_{20}p_{11} + 2p_{20}c_1 - p_{20} + 2p_{11}c_1 - p_{11} - c_1, \\ & 2c_2 + s_2 - p_{20} - p_{11} - c_1 \}. \end{aligned}$$

We eliminate the first three colored polynomials containing variables of Z and derive $\langle H \rangle = \langle G_A \rangle + \langle H_Y \rangle$ with $H_Y = H_B \cap \mathbb{Q}[Y]$.

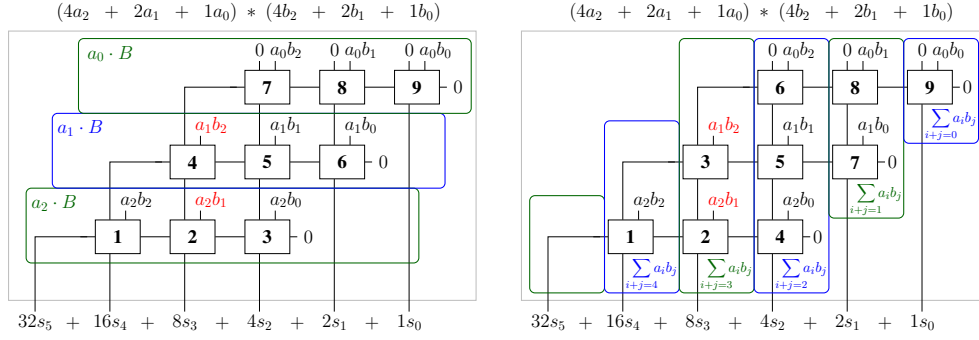


Figure A.4: Standard row-wise slicing (left) and our column-wise slicing (right) for a clean 3-bit carry-save-adder based (CSA) multiplier. The numbers in the full-adders depict the order.

A.6 Order

As long as the gate and field polynomials are ordered according to a reverse topological lexicographic term order, the choice of order does not affect the correctness of the approach, cf. Thm. A.15. However the choice of order has an influence on the number of monomials in the intermediate reduction result [103]. Hence, in addition to rewriting and reducing the Gröbner basis G , choosing an appropriate term and hence reduction order has a tremendous effect on computation time.

Given the two-dimensional structure of multipliers, two orderings seem well fitted, namely a row-wise and a column-wise ordering. The idea in both approaches is to partition the gates of a circuit into *slices*, which are then totally ordered. The gates within a slice are ordered reverse topologically. The combined order of the variables has to be reverse topological, such that the requirements of Thm. A.15 are fulfilled and hence the gate and input field polynomials form a Gröbner basis.

In the row-wise approach the gates are ordered according to their backward level. The ordering is abstractly depicted in the left circuit in Fig. A.4, where the order of the full-adders in a clean carry-save-adder based (CSA) multiplier is given. Informally, a multiplier is *clean* when neither gate synthesis nor mapping is applied and where the XOR-gates, partial products and the half/full adders can easily be identified. Otherwise a multiplier is called *dirty*. In previous work the row-wise approach is widely used. In the approach of [103] the gates are ordered according to their logic level based on the circuit inputs. In [29] the row-wise order is used to derive a word-level specification for a CSA step in a clean CSA multiplier. Unfortunately, the variable order is only roughly discussed in [93].

In the column-wise order, cf. right side of Fig. A.4, the multiplier is partitioned vertically such that each slice contains exactly one output bit. We will use this order to determine a more robust incremental checking approach.

In Fig. A.4 we also list the sum of the partial products which occur in the row-wise and column-wise slices. Assume we swap $a_1 b_2$ and $a_2 b_1$. In contrast to permuting

partial products within a row, permuting partial products within a column does not affect the correctness of the multiplier. By exchanging a_1b_2 and a_2b_1 the given sums of partial products for the row-wise slices are not valid anymore, whereas in the column-wise slicing the sum of partial products is still correct, meaning we can uniquely identify the partial products in a column-wise slice.

A.7 Incremental Column-Wise Checking

The goal of an incremental checking algorithm is to divide the verification problem into smaller, less complex and thus more manageable sub-problems. Because a column-wise term order is robust under permutation of partial products, we use such an order to define our incremental slices. Furthermore we split the word-level specification of Def. A.13 into smaller specifications which relate the partial products, incoming carries, sum output bit and the outgoing carries of each slice.

Definition A.32. Let C be a circuit which is partitioned according to a column-wise term order, such that each slice contains exactly one output bit. For column i with $0 \leq i < 2n$ let $P_i = \sum_{k+l=i} a_k b_l$ be the *partial product sum* (of column i).

Definition A.33. Let C be a circuit, as defined in Sect. A.3. A sequence of $2n + 1$ polynomials C_0, \dots, C_{2n} over the variables of C is called a *carry sequence* if

$$-C_i + 2C_{i+1} + s_i - P_i \in I(C) \quad \text{for all } 0 \leq i < 2n + 1$$

Then the $R_i = -C_i + 2C_{i+1} + s_i - P_i$ polynomials are called the *carry recurrence relations* for the sequence C_0, \dots, C_{2n} .

Based on these definitions we can obtain a general theorem which allows to incrementally verify multiplier circuits using carry recurrence relations. For this theorem it is not necessary to know how the carry sequence is actually derived.

Theorem A.34. Let C be a circuit where all carry recurrence relations are contained in $I(C)$, i.e., C_0, \dots, C_{2n} define a carry sequence as in Def. A.33. Then C is a multiplier in the sense of Def. A.13, if and only if $C_0 - 2^{2n}C_{2n} \in I(C)$.

Proof. By the condition of Def. A.33, we have (modulo $I(C)$)

$$\begin{aligned} \sum_{i=0}^{2n-1} 2^i s_i &= \sum_{i=0}^{2n-1} 2^i (P_i + C_i - 2C_{i+1}) \\ &= \sum_{i=0}^{2n-1} 2^i P_i + \underbrace{\sum_{i=0}^{2n-1} (2^i C_i - 2^{i+1} C_{i+1})}_{C_0 - 2^{2n} C_{2n}}. \end{aligned}$$

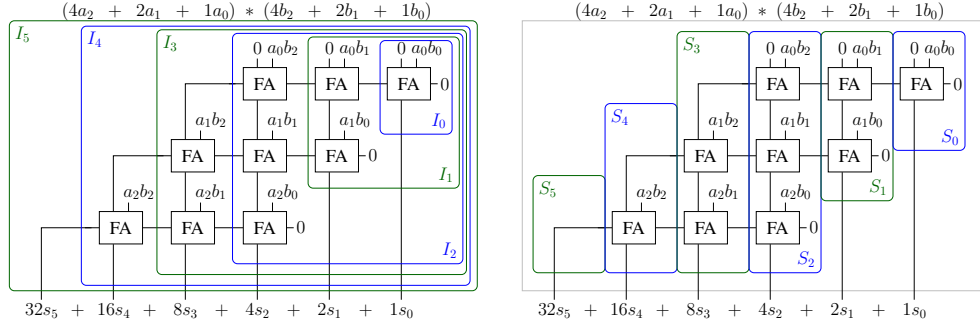


Figure A.5: Deriving input cones (left) and slices (right) for a clean 3-bit CSA multiplier.

It remains to show $\sum_{i=0}^{2n-1} 2^i P_i = (\sum_{i=0}^{n-1} 2^i a_i)(\sum_{i=0}^{n-1} 2^i b_i)$:

$$\sum_{i=0}^{2n-1} 2^i P_i = \sum_{i=0}^{2n-1} 2^i \sum_{\substack{k+l=i \\ k,l \geq 0}}^{k,l \leq n-1} a_k b_l = \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} 2^{k+l} a_k b_l = \left(\sum_{k=0}^{n-1} 2^k a_k \right) \left(\sum_{l=0}^{n-1} 2^l b_l \right)$$

Putting the above calculations together yields:

$$\underbrace{\sum_{i=0}^{2n-1} 2^i s_i}_L = \underbrace{(C_0 - 2^{2n} C_{2n})}_{L_1} + \underbrace{\left(\sum_{k=0}^{n-1} 2^k a_k \right) \left(\sum_{l=0}^{n-1} 2^l b_l \right)}_{L_2}$$

Since all $R_i \in I(C)$, it holds that $L - L_1 - L_2 \in I(C)$. For soundness, we assume $L_1 \in I(C)$, thus conclude $L - L_2 \in I(C)$, which proves that C is a multiplier. For completeness, let $L - L_2 \in I(C)$ and thus $L_1 \in I(C)$. \square

For our incremental checking algorithm we determine for each output bit s_i its input cone, namely the gates which s_i depends on (cf. left side of Fig. A.5):

$$I_i := \{\text{gate } g \mid g \text{ is in input cone of output } s_i\}$$

We derive slices S_i as the difference of consecutive cones I_i (cf. right side of Fig. A.5):

$$S_0 := I_0 \quad S_{i+1} := I_{i+1} \setminus \bigcup_{j=0}^i S_j$$

Definition A.35 (Sliced Gröbner Bases). Let G_i be the set of polynomial representations of the gates in a slice S_i , cf. Eqn. (A.1), and the input field polynomials. The terms are ordered such that the requirements of Thm. A.15 are fulfilled.

Corollary A.36. The set G_i is a Gröbner basis for the slice ideal $\langle G_i \rangle$.

Proof. Follows from Thm. A.15 with C replaced by S_i and G replaced by G_i . \square

Algorithm 2: Multiplier Checking Algorithm**Input** : Circuit C with sliced Gröbner bases G_i **Output** : Determine whether C is a correct multiplier

```

1  $C_{2n} \leftarrow 0$ ;
2 for  $i \leftarrow 2n - 1$  to 0 do
3    $C_i \leftarrow \text{Remainder}(2C_{i+1} + s_i - P_i, G_i \cup F_i)$ 
4 return  $C_0 = 0$ 

```

Since the ideal $\langle G_i \rangle$ contains all the field polynomials F_i for the gate variables in S_i , we may use them in the reduction process to eliminate exponents greater than 1 in the intermediate reduction results. Our incremental checking algorithm, cf. Alg. 2, works as follows: We start at the last output bit s_{2n-1} and compute the polynomials C_i recursively as the remainder of dividing $2C_{i+1} + s_i - P_i$ by $G_i \cup F_i$. Hence a polynomial C_i is uniquely defined, given P_i and C_{i+1} . It remains to fix the boundary polynomial C_{2n} , where we simply choose $C_{2n} = 0$.

Theorem A.37. *Algorithm 2 returns true iff C is a multiplier.*

Proof. By definition $R_i := -C_i + 2C_{i+1} + s_i - P_i \in \langle G_i \cup F_i \rangle$. Let F denote the set of all field polynomials for the variables of C . Since $G_i \subseteq G$ and $F_i \subseteq F$, we have $G_i \cup F_i \subseteq G \cup F$. Furthermore $\langle G \cup F \rangle = \langle G \rangle = J(C)$ and thus $R_i \in J(C) = I(C)$.

We show inductively that C_i is reduced w.r.t. U_i , where $U_i := \bigcup_{j \geq i} (G_j \cup F_j)$. This requires that s_i and P_i are reduced w.r.t. to U_{i+1} , which holds due to the construction of the sliced Gröbner bases G_i . By $U_0 = G \cup F$ we can derive that the final remainder C_0 is reduced w.r.t. $G \cup F$ thus $C_0 = C_0 - 2^{2n}C_{2n} \in I(C) = J(C)$ iff $C_0 = 0$, which concludes the proof using Thm. A.34. \square

Consequently Alg. 2 returns *false* iff a multiplier is incorrect, i.e., $C_0 \neq 0$. As discussed in the final remark of Sect. A.3 we can use C_0 to receive a concrete counterexample. It also is possible to abort the algorithm earlier if we find partial products $a_k b_l$ of higher slices $S_{k+l} = S_j$ in remainders C_i with $i < j$.

A.8 Incremental Equivalence Checking

In this section we introduce an incremental equivalence checking algorithm [91] generalizing our incremental checking approach to gate-level equivalence checking of two multipliers, but the approach is not restricted to multiplier circuits only. The presented theory applies to all acyclic circuits C, C' which have the same inputs and the same number of output bits. We generalize our definition of circuits of Sect. A.3 as follows.

Let C be a circuit with l boolean inputs a_0, \dots, a_{l-1} and m output variables s_0, \dots, s_{m-1} . Internal gates are represented by g_0, \dots, g_j . Further let C' be a circuit with the same l boolean inputs, but m different outputs s'_0, \dots, s'_{m-1} . The gates of C'

are defined by gate variables g'_0, \dots, g'_k . The union of C, C' is denoted by $C \cup C'$, for which we can determine $I(C \cup C') = J(C \cup C')$ as described in Sect. A.3.

The core idea of equivalence checking is to verify that two circuits C and C' compute the same output, given the same input. The benefit of equivalence checking is that a circuit can be verified without requiring a word-level specification by checking the equivalence of a circuit and a correct “golden” reference circuit. In the following we show how we can derive an incremental equivalence checking approach based on our column-wise checking algorithm of Sect. A.7.

Definition A.38. Let C, C' be two circuits. They are *equivalent*, written $C \equiv C'$, if

$$s_i - s'_i \in I(C \cup C') \quad i = 0, \dots, m-1.$$

Lemma A.39.

$$C \equiv C' \quad \text{iff} \quad \sum_{i=0}^{m-1} 2^i (s_i - s'_i) \in I(C \cup C')$$

Proof. “ \Rightarrow ”: Follows from Def. A.2.

“ \Leftarrow ”: Let $\varphi: X \rightarrow \mathbb{B} \subseteq \mathbb{Q}$ denote an evaluation of all variables X of C, C' , which is implied by the functionality of the circuit gates, e.g., values of s_i, s'_i in \mathbb{B} are uniquely determined given fixed inputs a_0, \dots, a_{l-1} . We extend φ to an evaluation of polynomials in the natural way (the unique homomorphic extension), i.e., $\varphi: \mathbb{Q}[X] \rightarrow \mathbb{Q}$. For all PCCs f , i.e. $f \in I(C \cup C')$, it holds by definition that $\varphi(f) = 0$. Since $\varphi(s_i), \varphi(s'_i) \in \mathbb{B}$ it is clear that $\varphi(s_i - s'_i) \in \{-1, 0, 1\}$.

Assume $\sum_{i=0}^{m-1} 2^i (s_i - s'_i) \in I(C \cup C')$, but $C \not\equiv C'$. Then there is a largest k with $0 \leq k < m$ and $\varphi(s_k - s'_k) \neq 0$, which gives the following contradiction

$$\begin{aligned} 0 &= \varphi\left(\sum_{i=0}^{m-1} 2^i (s_i - s'_i)\right) = \sum_{i=0}^k 2^i \varphi(s_i - s'_i) \\ &= \underbrace{2^k \varphi(s_k - s'_k)}_{\in \{-2^k, 2^k\}} + \underbrace{\sum_{i=0}^{k-1} 2^i \varphi(s_i - s'_i)}_{\in [-2^k+1, 2^k-1]} \neq 0 \end{aligned}$$

□

As for the incremental checking algorithm we define a sequence of relations, which is used to split the word-level equivalence specification. Based on the sequence we define an abstract incremental bit-level equivalence checking algorithm.

Definition A.40. Let C, C' be two circuits. A sequence of m polynomials $\Delta_0, \dots, \Delta_m$ over the variables of C, C' is called a sequence of *slice polynomials* if

$$-\Delta_i + 2\Delta_{i+1} + (s_i - s'_i) \in I(C \cup C') \quad \text{for all } 0 \leq i < m$$

The polynomials $E_i = -\Delta_i + 2\Delta_{i+1} + (s_i - s'_i)$ are called *slice relations* for the sequence $\Delta_0, \dots, \Delta_m$.

Theorem A.41. Let C, C' be two circuits and $\Delta_0, \dots, \Delta_m$ be a sequence of slice polynomials. Then $C \equiv C'$ in the sense of Def. A.38 iff $2^m \Delta_m - \Delta_0 \in I(C \cup C')$.

Proof. Using Def. A.40 we obtain modulo $I(C \cup C')$

$$\sum_{i=0}^{m-1} 2^i (s_i - s'_i) = \sum_{i=0}^{m-1} 2^i (2\Delta_{i+1} - \Delta_i) = 2^m \Delta_m - \Delta_0. \quad \square$$

Before we can define our incremental equivalence checking algorithm, we need to find a Gröbner basis for the ideal $I(C \cup C')$ and similar to Sect. A.7 we will define input cones which are then used to define slices S_i .

Lemma A.42. Let C and C' be two circuits. Let G, G' be Gröbner bases for $I(C), I(C')$ w.r.t. \leq, \leq' , satisfying the conditions of Thm. A.15. Further let \leq_{\cup} be such that \leq, \leq' are contained in \leq_{\cup} . Then $G \cup G'$ is a Gröbner basis for $I(C \cup C')$ w.r.t. \leq_{\cup} .

Proof. The set $G \cup G'$ consists of all gate polynomials of C, C' and input field polynomials $a_i(a_i - 1)$, but no more. Since all variables of C, C' apart from the input variables are unequal, $G \cap G'$ contains only the input field polynomials.

Since the variables a_0, \dots, a_{l-1} are the smallest elements in \leq, \leq' they are by definition also the smallest elements in \leq_{\cup} . Furthermore the term orderings for the gate polynomials of C and C' are still valid in \leq_{\cup} . Hence by the constraints on \leq_{\cup} the leading term of a polynomial in $G \cup G'$ is either the output variable of a circuit gate or the square of an input variable. Thus by Lemma A.6 $G \cup G'$ is a Gröbner basis for $I(C \cup C')$ w.r.t. \leq_{\cup} . \square

For each pair of output bits s_i and s'_i we determine its input cone

$$I_i := \{\text{gate } g \mid g \text{ is in input cone of output } s_i \text{ or } s'_i\}.$$

The slices S_i are defined as in Sect. A.7 as difference of consecutive cones I_i . For each slice we define a set of polynomials G_i according to Def. A.35. By Cor. A.36 such a set is a Gröbner basis for the ideal generated by the input field polynomials and the gate polynomials of a slice. Note that the ideal generated by G_i contains all the field polynomials F_i for the gate variables in S_i .

Using Thm. A.41 we define our incremental equivalence checking algorithm, cf. Alg 3. Setting the boundary $2^m \Delta_m$ to 0 we obtain the sequence of slice polynomials $\Delta_0, \dots, \Delta_{m-1}$ recursively by computing each Δ_i as the remainder of $2\Delta_{i+1} + s_i - s'_i$ modulo the sliced Gröbner bases $G_i \cup F_i$. This ensures that all E_i are contained in $\langle G_i \cup F_i \rangle \subseteq I(C \cup C')$. After computing $\Delta_0, \dots, \Delta_{m-1}$ we have to check if $\Delta_0 = 0$.

By similar arguments as in the proof of Thm. A.37 we show correctness of Alg 3.

Theorem A.43. Algorithm 3 returns true iff $(C \equiv C')$.

Proof. It holds by definition that $E_i = -\Delta_i + 2\Delta_{i+1} + (s_i - s'_i) \in \langle G_i \cup F_i \rangle$. By F we denote the set of all field polynomials of the variables of C, C' . We can derive that $G_i \cup F_i \subseteq G \cup G' \cup F$ Therefore $E_i \in \langle G \cup G' \cup F \rangle = I(C \cup C')$.

Algorithm 3: Equivalence Checking Algorithm

Input : Circuits C, C' with sliced Gröbner bases G_i
Output : Decide if C and C' are equivalent ($C \equiv C'$)

```

1  $\Delta_m \leftarrow 0$ ;
2 for  $i \leftarrow m - 1$  to 0 do
3    $\Delta_i \leftarrow \text{Remainder}(2\Delta_{i+1} + s_i - s'_i, G_i \cup F_i)$ 
4 return  $\Delta_0 = 0$ 

```

Algorithm 4: Outline of our tool AIGMULTOPOLY

Input : Circuit in AIG format
Output : File f for computer algebra system

```

1 for  $i \leftarrow 0$  to  $2n - 1$  do
2    $S_i \leftarrow \text{Define-Cones-of-Influence}(i)$ ;
3    $\text{Merge}(S_i)$ ;
4    $\text{Promote}(S_i)$ ;
5    $\text{Levelize}(S_i)$ ;
6    $\text{Apply-Rewriting}(S_i)$ ;
7    $\text{Identify-Vanishing Constraints}(S_i)$ ;
8  $f \leftarrow \text{Print to file}$ ;

```

We show inductively that Δ_i is reduced w.r.t. $U_i := \bigcup_{j \geq i} (G_j \cup F_j)$. For the induction it is required that s_i and s'_i are reduced w.r.t. to U_{i+1} , which holds due to the definition of the sliced Gröbner bases. With $U_0 = G \cup G' \cup F$ we get Δ_0 is reduced w.r.t. $G \cup G' \cup F$ thus $\Delta_0 = 2^m \Delta_m - \Delta_0 \in J(C \cup C')$ iff $\Delta_0 = 0$, concluding the proof using Thm. A.41. \square

A.9 Engineering

In this section we present the outline of our tool AIGMULTOPOLY [89], cf. Alg. 4, and present a novel approach to define column-wise slices. Our tool AIGMULTOPOLY, which is implemented in *C*, takes a circuit given as an AIG in AIGER format [16] as input and returns a file which can be passed on to the computer algebra systems Mathematica [102] and Singular [38].

In AIGMULTOPOLY we define the cones-of-influence, which are used to define the column-wise slices. In certain cases we have to optimize the slices by moving nodes from one slice to another slice, which we discuss further down. After slicing an ordering is defined for the nodes inside a slice, the rewriting methods are applied and as a last step everything including the computation instructions of our incremental column-wise verification algorithm in the syntax of the computer algebra system is printed to a file. In the computer algebra system the actual computation (repeated multivariate division) of the incremental checking algorithm is executed.

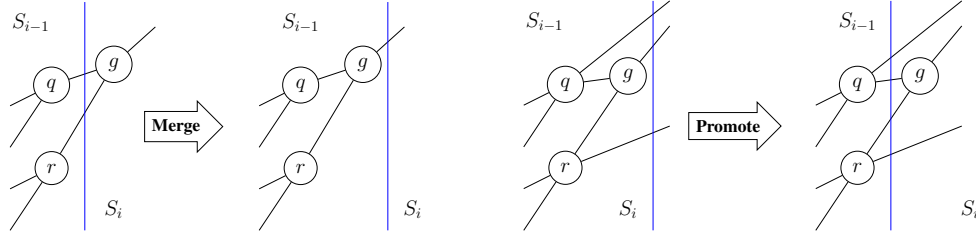


Figure A.6: Moving nodes between slices by Merge (left side) and Promote (right side).

We generally define the column-wise slices based on the input cones, cf. Sect. A.7. But this is not always precise enough for dirty multipliers. It frequently happens that AIG-nodes which are not directly used to compute the output s_i of a slice are allocated to later slices. This happens for example for carry outputs of full- and half-adders when they do not share their nodes with the sum output.

Example A.44. In Fig. A.2 the dashed lines depict an optimal column-wise slicing. If we would define the slices only based on input cones, then the AND-gate with output c_1 would belong to S_2 . Similar for the gates with outputs c_2, g_2, g_1, c_3 , thus all the full- and half-adders would be cut into two pieces.

We want to have these nodes in the same slice as the nodes computing the sum output of an adder. Otherwise we cannot apply Adder-Rewriting. We informally define those nodes in a slice S_i which are used as inputs of nodes in a slice S_j with $j > i$ as *carries of a slice S_i* . The size of the carry polynomial C_i can be reduced by decreasing the number of carries of the corresponding slice S_i . If the nodes are not moved, larger carry polynomials C_i are generated and hence we get larger intermediate reduction results than necessary. Therefore we eagerly move nodes between slices in a kind of peephole optimization, backward (*Merge*) as well as forward (*Promote*).

Merge Assume we find a node g in the AIG which belongs to a slice S_i and both children q and r belong to smaller slices S_j and S_k . Let $l = \max(j, k)$. If the children q and r do not have any other parent than g in a bigger slice than S_l , we move the node g back to slice S_l . This approach is depicted on the left side of Fig. A.6 for $j = k = i - 1$. Thus after merging g , the nodes q, r are less likely to be carry variables any more, especially when $j = k$. We apply merging repeatedly until completion and S_l and S_i are updated after each application. Merging nodes usually ensures that the complete logic of a full- or half-adder is contained within one slice.

Example A.45. In the circuit of Fig. A.2 gate c_1 is merged to slice S_1 . Gates g_1, g_2, c_2 are repeatedly merged to S_2 and gate c_3 is merged from S_4 to S_3 . Hence every full- or half-adder logic is contained within one slice.

Promote In some multiplier architectures it happens the inputs of a node g are contained in the same slice and all three nodes are carries. In this case

we decrease the number of carries by promoting g to the next bigger slice. More precisely we search for nodes g in a slice S_{i-1} which have exactly one parent contained in a larger slice S_j with $j \geq i - 1$. If g would also be an input of a node in S_{i-1} , we cannot move g to slice S_i without violating the topological order. The inputs of g also have to be contained in S_{i-1} and need to have at least one parent in a bigger slice S_j with $j > i - 1$, i.e., they are carries. Then we promote g to slice S_i and thus decrease the number of carries. Promoting is shown on the right side of Fig. A.6 for $j = i$.

A node g which is merged cannot be promoted back in the next round, because merging and promoting have different requirements for the input nodes of g . This prevents an endless alternate application of the above rules.

We can overcome the necessity of merging gates by defining slices based on the output cones of the partial products, i.e., gates which depend on a partial product. This approach works only if the partial products are generated by a simple AND-gate. If for example Booth encoding of partial products is applied, we cannot identify all partial products in the AIG and thus cannot apply the approach of defining slices based on the output cones.

$$O_i := \{\text{gate } g \mid g \text{ is in output cone of a partial product } a_k b_l \text{ with } k + l = i\}$$

We derive slices S_i as the difference of consecutive cones O_i :

$$S_{n-2} := O_{n-2} \quad S_i := O_i \setminus \bigcup_{j=i+1}^{n-2} S_j$$

The disadvantage of this approach is that the slice S_{n-2} actually contains two output bits, namely s_{n-2} and s_{n-1} . In an AIG the output bit is usually introduced by a relation of the form $s = g_k$, i.e., renaming of a gate variable g_k . To solve the issue we simply define a slice S_{n-1} which contains exactly the relation $s_{n-1} = g_k$ for some g_k . This constraint is removed from S_{n-2} .

It can be seen in Fig. A.6 that slicing based on the output cones makes the concept of merging superfluous. The node g in slice S_i has inputs q and r , which belong to smaller slices S_j and S_k . Hence g depends on the partial products of q and r . Thus g is in the same output cone than its children and it will be allocated to S_l , with $l = \max(j, k)$. So it cannot belong to a different slice.

In contrast to merging, promoting a node is still necessary, because as it can be seen in the right side of Fig. A.6, nodes g, q, r all depend on the same partial products, hence they will all be allocated to S_{i-1} , which makes promoting of g to S_i still necessary. Since promoting is necessary in both approaches and slicing based on the input cones also works for encodings, such as Booth encoding, we will stick to the slicing based on input cones.

After merging and promoting, the allocation of nodes to a slice is fixed. The slices are totally ordered starting from S_{2n-1} to S_0 . We order the nodes in a slice according to

their level seen from the circuit inputs. Ordering the nodes after merging and slicing ensures that the variables are topologically sorted.

The rewriting techniques of Sect. A.4 are applied in the order: Adder-Rewriting, XOR-Rewriting and Common-Rewriting. Since the structures of full- and half-adders usually do not change within a certain circuit, we do not have to compute the Gröbner basis H_B , cf. Sect. A.5, every time we find a certain full- or half-adder structure in the corresponding AIG. The polynomials describing the adder will always have the same form. Thus it suffices that we know the structure of the polynomials in H_B and simply replace the polynomials describing the adder structure by the polynomials of H_B with appropriate variables. The same applies to structures describing an XOR- or XNOR-gate.

In order to simulate Common-Rewriting, we search in each slice S_i for nodes which are not used in another slice and have exactly one parent. We collect them in the set U_i . Polynomials of nodes in S_i which depend on nodes in U_i are reduced first by the polynomials of nodes in U_i , thus eliminating the nodes of U_i .

After rewriting S_i , we search for Vanishing Constraints in the remaining nodes of S_i . More precisely we search for products which always evaluate to zero, e.g., gb in Example A.8. We store these constraints in a set V_i and during remainder computation we also reduce against elements of V_i . Since these constraints are contained in the ideal $I(C)$, and because of Thm. A.16, we can add these polynomials to the Gröbner basis without violating the Gröbner basis property.

Partial Product Elimination is handled internally. We search for all n^2 nodes which define a partial product in the AIG and check if they are correct. We exclude the original inputs from the AIG and treat these nodes as new inputs of the AIG. In the printing process we simply rewrite the specification in terms of these nodes.

The polynomials of each slice together with computation instructions for the incremental checking algorithm are written to a file which can be passed on to the computer algebra systems Mathematica or Singular. Whereas Singular treats the polynomials of the sliced Gröbner bases as a set which is then ordered internally according to the given variable order, it seems that Mathematica actually treats the set of polynomials as a list. Therefore it is necessary to print the polynomials in the correct order. We did not obey this fact in [89], where we actually printed the polynomials in reverse order. We started by printing the polynomials defining the partial products and ended by printing the polynomial representation of the output bit of each slice. By adjusting the printing order of the polynomials such that the leading terms of the polynomials are ordered according to the given variable order we were able to improve our computation results from [89].

A.10 Experiments

In our work we focus on integer multipliers, as the authors of [32, 93, 94, 103], which take two n -bit vectors as inputs and return a bit-vector of size $2n$. In the work of [32, 103] the authors used clean CSA multipliers, crafted from [69]. They further used multipliers

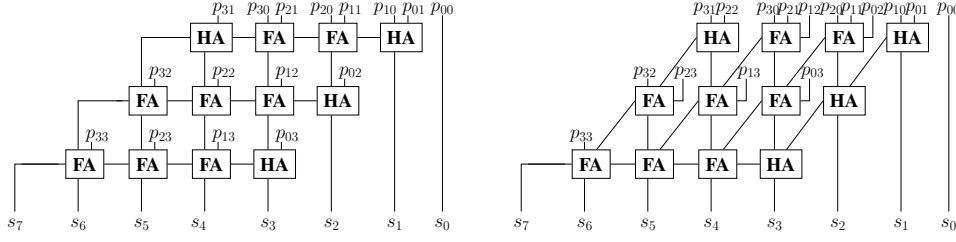


Figure A.7: Multiplier architectures of “btor” (left) and “sp-ar-rc” (right) for input bit-width $n = 4$.

generated by ABC [10] on which synthesis and technology mapping is applied. These multipliers are extremely hard to verify [32, 103].

In our experiments we focus on two different architectures, called “btor” and “sp-ar-rc”. The “btor” benchmarks are generated from Boolector [83] and can be considered as clean multipliers. The “sp-ar-rc” multipliers are part of the bigger AOKI benchmarks [53] and can be considered as dirty multipliers. The AOKI benchmark set was used extensively in the experiments of [93, 94]. The structure of “btor” and “sp-ar-rc” multipliers is shown in Fig. A.7. Both architectures can be fully decomposed into full- and half-adders, which are then accumulated. In “btor” these full- and half-adders are accumulated in a grid-like structure, whereas in “sp-ar-rc” full- and half-adders are accumulated diagonally.

In addition to “btor” and “sp-ar-rc” multipliers, we will further use more complex multiplier architectures of [32, 103] and of the AOKI benchmarks. The architectures of the different AOKI benchmarks are indicated by the names of the multipliers. The naming of the multipliers follows the following structure: “partial product generation - accumulation - last step adder”, e.g., a “sp-ar-rc” multiplier consists of simple partial product generation, which are accumulated in an array structure and the adder in the last accumulation step is a ripple-carry adder. In our experiments we will include “bp-ar-rc”, “sp-ar-cl” and “sp-wt-rc”, where *bp* defines booth encoding [85], *cl* defines a carry look-ahead adder and *wt* means accumulation by a Wallace-tree structure, where the number of partial products is reduced as soon as possible, which minimizes the overall delay of the multiplier [85].

Furthermore we use benchmarks which are synthesized and technology mapping is applied. The basis of these benchmarks is an “abc”-multiplier, which is generated with ABC [10] and has the same clean structure as the “btor” benchmarks. The different versions of synthesis and technology mapping should be the same as in [32, 103].

In all our experiments we used a standard Ubuntu 16.04 Desktop machine with Intel i7-2600 3.40GHz CPU and 16 GB of main memory. The (wall-clock) time limit was set to 1200 seconds and main memory was limited to 14GB. We measure the time from starting our tool AIGMULTOPOLY until Mathematica resp. Singular are finished. This includes the time our tool AIGMULTOPOLY needs to generate the files for the computer algebra system, which in the worst case is around 3 seconds for 128-bit multipliers. The results also include the time to launch Mathematica resp. Singular. We mark unfinished

mult	n	Mathematica			Singular		
		+inc	-inc +col	-inc +row	+inc	-inc +col	-inc +row
btor	16	2	3	4	1	1	2
btor	32	14	56	106	10	42	42
btor	64	131	MO	MO	MO	MO	MO
btor	128	TO	TO	TO	EE	EE	EE
sp-ar-rc	16	4	9	11	1	TO	TO
sp-ar-rc	32	30	326	425	28	TO	TO
sp-ar-rc	64	300	MO	MO	MO	MO	MO
sp-ar-rc	128	TO	TO	TO	EE	EE	EE

Table A.1: Our column-wise incremental approach (+inc +col) vs. a non-incremental approach using column-wise (-inc +col) and row-wise order (-inc +row) without Adder-Rewriting.

experiments by TO (reached the time limit), MO (reached the memory limit) or by an error state EE (reached the maximum number of ring variables in Singular). Singular has a limit of 32767 on the number of ring variables and multipliers of larger bit-width easily exceed this limitation. We also mark some unfinished experiments by TO*, in this case the time limit was set to 36000 seconds (10 hours). Experimental data, benchmarks and source code is available at <http://fmv.jku.at/fmsd18>.

In Table A.1 we compare our incremental column-wise verification approach of Alg. 2 to a non-incremental verification approach, where the complete word-level specification (Def. A.13) is reduced. For the non-incremental approach we use a column-wise as well as row-wise term ordering. In Table A.1 all optimizations are enabled (XOR-Rewriting, Common-Rewriting, Vanishing Constraints, Merge, Promote), but Adder-Rewriting is disabled. The results show that our incremental verification approach is faster and uses less memory than the non-incremental approaches. In the experiments of [89] Mathematica needed a lot more time than Singular, but as discussed at the end of Sect. A.9 we could improve the running time of Mathematica by adjusting the printing order. Hence in the experiments presented in this work the computation time of Mathematica and Singular is nearly the same. The big difference between the two computer algebra systems is that Singular needs a lot of memory, verification of 64-bit multipliers needs more than 14 GB. As expected we get an error state for the 128-bit multipliers.

By default the adapted optimizations XOR-Rewriting, Common-Rewriting and adding Vanishing Constraints of [93] are enabled in our incremental column-wise checking algorithm. In the experiments shown in Table A.2 we show the effects of turning off exactly one of these optimizations (keeping Adder-Rewriting disabled). For the “btor” multipliers turning off Common-Rewriting actually speeds up computation time. In the “btor” multipliers only a few gates with only one parent exist and applying common-rewriting by splitting remainder computation increases the run-time. In “sp-ar-rc” multipliers turning off Common-Rewriting increases computation time drastically,

mult	n	Mathematica				Singular			
		+inc	-xor	-com	-vc	+inc	-xor	-com	-vc
btor	16	2	5	2	3	1	1	1	1
btor	32	14	31	4	15	10	28	5	12
btor	64	131	292	22	128	MO	MO	MO	MO
btor	128	TO	TO	186	TO	EE	EE	EE	EE
sp-ar-rc	16	4	17	TO	4	1	6	TO	1
sp-ar-rc	32	30	171	TO	31	28	242	TO	28
sp-ar-rc	64	300	TO	TO	303	MO	EE	MO	MO
sp-ar-rc	128	TO	TO	TO	TO	EE	EE	EE	EE

Table A.2: Effect of turning off optimizations XOR-Rewriting (-xor), Common-Rewriting (-com) and Vanishing Constraints (-vc) keeping Adder-Rewriting disabled.

mult	n	Mathematica				Singular			
		+inc	-merge	-prom	+ocone	+inc	-merge	-prom	+ocone
btor	16	2	3	2	3	1	1	1	1
btor	32	14	21	15	15	10	10	10	11
btor	64	131	233	133	132	MO	MO	MO	MO
btor	128	TO	TO	TO	TO	EE	EE	EE	EE
sp-ar-rc	16	4	4	TO	4	1	1	TO	1
sp-ar-rc	32	30	39	TO	31	28	29	MO	28
sp-ar-rc	64	300	430	TO	301	MO	MO	MO	MO
sp-ar-rc	128	TO	TO	TO	TO	EE	EE	EE	EE

Table A.3: Effect turning off Merge (-merge) and Promote (-prom). Furthermore the effect of using slicing based on the output cones (+ocone).

because structures containing nodes with only one parent occur much more frequently. Turning off XOR-Rewriting is a downgrade for both clean and dirty multipliers. Because of the additional number of gates we already reach an error state for a 64-bit multiplier in Singular. In [89] turning off Vanishing Constraints had a very bad effect for clean multipliers in Mathematica. By printing the polynomials in a different order we could overcome this issue. Now turning off Vanishing Constraints does not influence the behavior of neither Mathematica nor Singular for clean as well as dirty multipliers. Hence the question can be asked if adding Vanishing Constraints in the current form is really necessary. Summarized it can be said that the optimizations have both positive and negative effects, depending on the structure of the multiplier.

In the experiments shown in Table A.3 we investigate the effects of turning off the engineering methods Merge and Promote. The computation time of disabling Merge can be considered to be the same. The difference can be seen in the size of C_i in the log-files, e.g., in sp-ar-rc-8 the maximum number of monomials in any C_i is 38, whereas

mult	n	Mathematica						Singular				
		+inc			+Adder Rew.			+inc		+Adder Rew.		
			+as		+ppe	-s		+as		+ppe	-s	
btor	16	2	2	1	1	0	1	0	1	0	0	
btor	32	14	15	2	2	2	10	11	1	1	1	
btor	64	131	132	11	6	5	MO	MO	14	9	5	
btor	128	TO	TO	101	47	40	EE	EE	EE	EE	EE	
sp-ar-rc	16	4	4	1	1	1	1	1	0	0	0	
sp-ar-rc	32	30	30	2	2	1	28	28	2	1	1	
sp-ar-rc	64	300	295	11	6	5	MO	MO	16	10	5	
sp-ar-rc	128	TO	TO	102	48	41	EE	EE	EE	EE	EE	

Table A.4: Enabling Adder-Rewriting and Partial Product Elimination.

in the approach with Merge enabled the maximum number is 8. Furthermore all C_i are linear. Turning off Promote does not affect “btor”-multipliers but really slows down computation time of “sp-ar-rc” multipliers. Furthermore we compare our incremental slicing based on the input cones to the slicing method which is based on the output cones. Both slicing approaches lead to identical output files for the computer algebra systems, hence we have the same computation time in both approaches.

In Table A.4 we apply Adder-Rewriting on top of our incremental verification approach. In the first step we simply add the full- and half-adder specifications (+as) to the Gröbner basis, without eliminating any internal variable. Comparing the computation time, it seems that computer algebra systems cannot use this additional redundant information, similar to Vanishing Constraints in Table A.2. Applying Adder-Rewriting by eliminating internal variables in the sliced Gröbner bases has a tremendous effect on the computation time. Now also 128-bit multipliers can be verified within roughly 100 seconds, while before verification timed out after 20 minutes. Additionally eliminating the partial products (+ppe) further speeds-up computation time. We assume that the considered multipliers are correct and since they can fully be decomposed into full- and half-adders, we never have to reduce by the sum output of a full- or half-adder separately. It is always reduced in parallel with the carry output. Elimination of the polynomials where the leading term is a sum-output of an adder from the Gröbner basis (-s) brings further improvements, but loses completeness.

In the experiments shown in Table A.5 we consider the more complex multiplier architectures introduced at the beginning of this section. We apply our default incremental-checking approach without Adder-Rewriting, because usually the regular full- and half-adder structures are destroyed by synthesis and technology mappings. Synthesizing and application of complex mappings makes it very hard to verify a circuit. Even an 8-bit multiplier cannot be verified any more, neither in Mathematica nor in Singular. This confirms the results of [32, 103]. It can further be seen that more complex architectures cannot be verified with the state-of-the-art approach, which makes more sophisticated

mult	n	Mathematica	Singular
abc-resyn3-no-comp	4	1	0
abc-resyn3-no-comp	8	2	7
abc-resyn3-no-comp	16	TO	TO
abc-resyn3-comp	4	1	0
abc-resyn3-comp	8	TO	TO
bp-ar-rc	4	TO	287
bp-ar-rc	8	TO	TO
sp-ar-cl	4	1	1
sp-ar-cl	8	TO	TO
sp-wt-rc	4	1	1
sp-wt-rc	8	2	1
sp-wt-rc	16	TO	TO

Table A.5: Complex multiplier architectures, including synthesis and technology mapping.

mult	n	Lingeling [13]	ABC [10]	-Adder Rew.	+Adder Rew.
btor vs. sp-ar-rc	8	14	12	2	1
btor vs. sp-ar-rc	16	TO*	TO*	6	1
btor vs. sp-ar-rc	32	-	-	44	3
btor vs. sp-ar-rc	64	-	-	443	15
btor vs. sp-ar-rc	128	-	-	TO	115

Table A.6: Incremental column-wise Equivalence checking with and without Adder-Rewriting.

reasoning necessary.

In the experiments of Table A.6 we apply the column-wise equivalence checking algorithm of Sect. A.8 and check the equivalence of the “btor” and “sp-ar-rc” multipliers. Despite their architectural similarities neither Lingeling [13] nor ABC [10] are able to verify their equivalence for $n = 16$ within 10 hours, whereas it takes only around a second using our approach based on computer algebra. In this experiment we only use Mathematica as a computer algebra system, because it supports more variables. We check the equivalence using our incremental equivalence checking algorithm with and without Adder-Rewriting. Enabling Adder-Rewriting again substantially reduces computation time. We do not use Partial Product Elimination, because in this setting we would have to manually map the AND-gates which generate the partial products of the two multipliers.

A.11 Conclusion

This article presents in detail our incremental column-wise verification approach to formally verify integer multiplier circuits, as introduced in [17, 89, 91].

We give a precise mathematical formalization of the theory of arithmetic circuit verification using computer algebra including rigorous soundness and completeness arguments. Our incremental column-wise checking algorithm has tremendously positive effects on computation time. We discuss several optimizations which rewrite and simplify the Gröbner basis. For these optimizations we introduce the necessary theory and present a technical theorem which allows us to rewrite local parts of the Gröbner basis based on [17]. Furthermore we show how our incremental verification algorithm can be extended to equivalence checking [91]. As a novel contribution we revise our engineering techniques and present a simple alternative method to define column-wise slices. We further improve computation times compared to [89] by changing the printing process of our tool.

As future work, we want to extend our methods to more complex architectures, i.e., we want to efficiently verify multiplier architectures used in Table A.5. We also want to consider overflow-semantics and negative numbers. Furthermore we want to investigate floating points and other word-level operators.

A.12 Acknowledgements

We would like to thank Paul Beame for sharing extended drafts of [7], then Mathias Soeken for helping to synthesize AOKI multipliers [53] used in their DATE'16 paper [93], Naofumi Homma for sending 128-bit versions of these benchmarks, Maciej Ciesielski for explaining the experimental set-up in [32, 103], the chairs and award committee of FMCAD'17 for selecting [89] as best paper and the invitation to contribute to this special issue, as well as the organizers of SYNASC'17 for the invited talk which lead to [17].



B

Paper B

A Practical Polynomial Calculus for Arithmetic Circuit Verification

Published In Proceedings of the 3rd Workshop on Satisfiability Checking and Symbolic Computation (SC'2) co-located with Federated Logic Conference (FLOC 2018), pages 61–76, Oxford, United Kingdom, 2018.

Authors Daniela Ritirc, Armin Biere and Manuel Kauers.

Acknowledgement This work is supported by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), Y464-N18, SFB F5004.

Abstract Generating and automatically checking proofs independently increases confidence in the results of automated reasoning tools. The use of computer algebra is an essential ingredient in recent substantial improvements to scale verification of arithmetic gate-level circuits, such as multipliers, to large bit-widths. There is also a large body of work on theoretical aspects of propositional algebraic proof systems in the proof complexity community starting with the seminal paper introducing the polynomial calculus. We show that the polynomial calculus provides a frame-work to define a practical algebraic calculus (PAC) proof format to capture low-level algebraic proofs needed in scalable gate-level verification of arithmetic circuits. We apply these techniques to generate proofs obtained as by-product of verifying gate-level multipliers using state-of-the-art techniques. Our experiments show that these proofs can be checked efficiently with independent tools.

B.1 Introduction

Formal verification gives correctness guarantees. However, the process of verification might also not be error-free. A common approach to increase confidence in the results of verification consists of generating machine checkable proofs which are then checked by

independent proof checkers. These checkers are less complex than for example theorem provers producing proofs and can also be verified.

For instance many applications of formal verification rely on SAT solvers. Their results can be validated by producing and checking resolution proofs [47, 107] or clausal proofs [43, 47]. Generating proofs is mandatory in the main track of the SAT Competition since 2016. These approaches have also recently been shown to scale to huge low-level proofs of combinatorial problems such as the Boolean Pythagorean triples problem [51] or Schur Number Five [46].

However, in certain applications, e.g., arithmetic circuit verification, resolution based SAT solving does not work. Especially reasoning about gate-level multipliers is considered to be hard [12]. For arithmetic circuit verification the currently most promising approach uses algebraic reasoning [32, 78, 89, 93].

In this approach each circuit gate is translated into a polynomial to model constraints between its output and inputs, i.e., roots of polynomials are identified as solutions of gate constraints. Additional polynomials ensure that values remain in the Boolean domain. Word-level specifications relating circuit outputs and inputs can also be translated into polynomials. Thus verification boils down to show that the specification polynomial is “implied” by the polynomials induced by the circuit gates (contained in the ideal generated by them).

To validate results of algebraic reasoning the polynomial calculus can be used [34]. It operates on polynomials and allows checking if a polynomial is a logical consequence of a given set of polynomials. The main focus in this area has been on proof complexity to obtain lower bounds for the degree and size of proofs [55]. For instance [82] introduces a general method to obtain lower bounds and [75] shows that certifying the non-k-colorability of graphs requires proofs of large degree. A more general calculus capable of detecting unsatisfiability of nonlinear equalities as well as inequalities is discussed in [96].

Our paper shows that the polynomial calculus can also be used in practice. In particular we generate low-level algebraic proofs needed to validate the results of ideal membership testing used in arithmetic circuit verification by translating proofs extracted from computer algebra systems to polynomial refutations in the polynomial calculus. After we review preliminaries in Sect. B.2, we present a concrete proof format for polynomial calculus proofs, called practical algebraic calculus in Sect. B.3. In Sect. B.4 we give a comprehensive introduction to arithmetic circuit verification, following [89]. Section B.5 introduces the tool flow of verifying and proof checking arithmetic circuits. In our experiments, shown in Sect. B.6, our new proof checker PACTRIM is used to independently validate the results of multiplier verification [89]. We further apply these techniques to equivalence checking of multipliers [91] and proving certain ring properties, e.g., commutativity of multipliers [7]. In general, we claim that our approach is the first to provide machine checkable proofs for current state-of-the-art techniques in verifying arithmetic circuits [32, 78, 89, 93].

B.2 Preliminaries

Proof systems are used to validate the results of verification systems. While a verification system only gives a *yes/no* answer, a proof system provides additionally a certificate with which the answer can be checked independently. We are concerned here with a proof system for reasoning about polynomial equations. The question is whether the zeroness of a certain set of polynomials implies the zeroness of another polynomial. We consider polynomials $p \in \mathbb{F}[X]$ where \mathbb{F} is a field and $X = \{x_1, \dots, x_n\}$ is a finite set of variables. The function $X \mapsto p(X)$ is called *polynomial function* of p . The *polynomial equation* of p is defined as $p(X) = 0$ and the solutions of this equation are the roots of p . From now on we drop the function argument and write $p = 0$ instead of $p(X) = 0$.

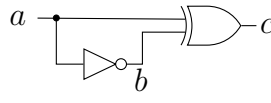
Reasoning with polynomial equations is well-understood both in computer algebra and in computational logic. Already Hilbert and collaborators have studied the theory of polynomial ideals in order to reason about the solution sets of polynomial equations. The application of Gröbner bases [25] by for instance Kapur [58, 59, 60] has turned the algebraic approach into a valuable computational tool for automated theorem proving with renewed recent interest [1, 108].

In order to introduce the notation and terminology needed later, let us give a brief summary of the theory. As far as algebra is concerned, we follow the standard textbooks [9, 26, 35]. From the logical perspective, we use a variant of the polynomial calculus (PC) as proposed by [34]. It is more flexible than the Nullstellensatz (NS) proof system [6], which is also heavily used in the proof complexity community. The relation between PC and NS in the context of our application is further discussed at the end of this section.

Let $G \subseteq \mathbb{F}[X]$ and $f \in \mathbb{F}[X]$. In logical terms, the question is whether the equation $f = 0$ can be deduced from the equations $g = 0$ with $g \in G$, i.e., every common root of the polynomials $g \in G$ is also a root of f . As we will only consider polynomial equations with right-hand side zero, we take the freedom to write f instead of $f = 0$. We write proofs as tuples $P = (p_1, \dots, p_n)$ of polynomials where each p_i is derived by one of the following rules.

Addition	$\frac{p_i \quad p_j}{p_i + p_j}$	p_i, p_j appearing earlier in the proof or are contained in G
Multiplication	$\frac{p_i}{qp_i}$	p_i appearing earlier in the proof or is contained in G and $q \in \mathbb{F}[X]$ being arbitrary

If f can be deduced from the polynomials $g \in G$, i.e., $p_n = f$, we write $G \vdash f$. In algebraic terms, $G \vdash f$ means that f belongs to the ideal generated by G . Recall that an *ideal* $I \subseteq \mathbb{F}[X]$ is defined as a set with $0 \in I$ and the closure properties $u, v \in I \Rightarrow u + v \in I$ and $w \in \mathbb{F}[X], u \in I \Rightarrow wu \in I$. If $G = \{g_1, \dots, g_m\} \subseteq \mathbb{F}[X]$ is a finite set of polynomials, then the ideal generated by G is defined as the set

$$G = \{ -b + 1 - a, \\ -c + a + b - 2ab, \\ a^2 - a, b^2 - b, c^2 - c \}$$


$$\frac{\frac{-c + a + b - 2ab}{-c + 1 - 2ab} \quad \frac{-b + 1 - a}{2ab - 2a + 2a^2}}{\frac{-c + 1 - 2a + 2a^2}{-c + 1}} \quad \frac{a^2 - a}{-2a^2 + 2a}$$

Figure B.1: The circuit, polynomial representation of the gates and proof for Ex. B.1.

$\{q_1g_1 + \dots + q_mg_m : q_1, \dots, q_m \in \mathbb{F}[X]\}$ and denoted by $\langle G \rangle$. The set G is called a *basis* of the ideal $\langle G \rangle$. It is clear that this is an ideal and that it consists of all the polynomials whose zeroness can be deduced from the zeroness of the polynomials in G . In logical terms we would call G an axiom system and $\langle G \rangle$ the corresponding theory. If we can derive $G \vdash 1$, or in algebraic terms $1 \in \langle G \rangle$, the PC proof is called a *PC refutation*.

Example B.1. This example shows that the output c of an XOR gate over an input a and its negation $b = \neg a$ is always true, i.e., $c = 1$ or equivalently $-c + 1 (= 0)$. We apply the polynomial calculus over the ring $\mathbb{Q}[c, b, a]$. Over \mathbb{Q} a NOT gate $x = \neg y$ is modeled by the polynomial $-x + 1 - y$ and an XOR gate $z = x \oplus y$ is modeled by the polynomial $-z + x + y - 2xy$. Because the variables are of the boolean domain we further need to enforce that every variable can only take the values 0 or 1. Therefore we add for each variable x_i a polynomial of the form $x_i(x_i - 1)$ to the given set of polynomials. The corresponding circuit representation, the given polynomials and a polynomial proof are shown in Fig. B.1.

Example B.2. Let $G = \{x, x + y\} \subseteq \mathbb{Q}[x, y]$, $f = y$. We have $G \vdash f$. A proof is $P = (-x, y)$. The first entry follows by the multiplication rule from x with $q = -1$, and the second entry follows by the addition rule from the first entry and $x + y$ which is contained in G .

Thanks to the theory of Gröbner bases [9, 25, 35], the polynomial calculus is decidable, i.e., there is an algorithm, which for any finite $G \subseteq \mathbb{F}[X]$ and $f \in \mathbb{F}[X]$ can decide whether $G \vdash f$ or not. A basis of an ideal I is called a Gröbner basis if it enjoys certain structural properties whose precise definition is not relevant for our purpose. What matters are the following fundamental facts:

- There is an algorithm (Buchberger's algorithm) which for any given finite set $B \subseteq \mathbb{F}[X]$ computes a Gröbner basis for the ideal $\langle B \rangle$ generated by B .
- Given a Gröbner basis G , there is a computable function $\text{red}_G : \mathbb{F}[X] \rightarrow \mathbb{F}[X]$ such that $\forall p \in \mathbb{F}[X] : \text{red}_G(p) = 0 \iff p \in \langle G \rangle$.

- Moreover, if $G = \{g_1, \dots, g_m\}$ is a Gröbner basis of an ideal I and $p, r \in \mathbb{F}[X]$ are such that $\text{red}_G(p) = r$, then there exist $h_1, \dots, h_m \in \mathbb{F}[X]$ such that $p - r = h_1g_1 + \dots + h_mg_m$, and such polynomials h_i can be computed.

Consider the extended calculus with the additional rule

$$\text{Radical} \quad \frac{p_i^m}{p_i} \quad m \in \mathbb{N} \setminus \{0\} \text{ and } p_i^m \text{ appearing earlier in the proof or is contained in } G.$$

If the polynomial f can be deduced from the polynomials g , where $g \in G$, with the rules of PC and this additional radical rule, we write $G \vdash^+ f$ and call this proof *radical proof* (\vdash^+). In algebra, the set $\{f \in \mathbb{F}[X] : G \vdash^+ f\}$ is called the *radical ideal* of G and is typically denoted by $\sqrt{\langle G \rangle}$.

Also the extended calculus \vdash^+ is decidable. It can be reduced to \vdash using the so-called Rabinowitsch trick [35, 4§2 Prop. 8], which says

$$f \in \sqrt{\langle G \rangle} \iff 1 \in \langle G \cup \{yf - 1\} \rangle \quad \text{or} \quad G \vdash^+ f \iff G \cup \{yf - 1\} \vdash 1,$$

depending whether you prefer algebraic or logic notation. In both cases, y is a new variable and the ideal/theory on the right-hand sides is understood as an ideal/theory of the extended ring $\mathbb{F}[X, y]$. The Rabinowitsch trick is therefore used to replace a radical proof (\vdash^+) by a PC refutation.

For a given set $G \subseteq \mathbb{F}[X]$, a *model* is a point $u = (u_1, \dots, u_n) \in \mathbb{F}^n$ such that for all $g \in G$ we conclude that $g(u_1, \dots, u_n) = 0$. Here, by $g(u_1, \dots, u_n)$ we mean the element of \mathbb{F} obtained by evaluating the polynomial g for $x_1 = u_1, \dots, x_n = u_n$. For a set $G \subseteq \mathbb{F}[X]$ and a polynomial $f \in \mathbb{F}[X]$, we write $G \models f$ if every model for G is also a model for $\{f\}$. Given $G \subseteq \mathbb{F}[X]$, define $V(G)$ as the set of all models of G . For an algebraically closed field \mathbb{F} , Hilbert's Nullstellensatz [35, 4§1 Thms. 1 and 2] asserts that $V(G)$ is nonempty if and only if $1 \notin \langle G \rangle$, and furthermore, $f \in \sqrt{\langle G \rangle} \iff V(G) \subseteq V(\{f\})$. In other words, $G \models f \iff G \vdash^+ f$. Particularly, the PC including the radical rule is correct (" \Leftarrow ") and complete (" \Rightarrow "). In combination with Rabinowitsch's trick, we can therefore decide the existence of models and furthermore produce certificates for the non-existence of models.

For our applications, only models $u \in \{0, 1\}^n \subseteq \mathbb{F}^n$ matter. Let us write $G \models_{\text{bool}} f$ if every model $u \in \{0, 1\}^n$ of G is also a model of $\{f\}$. Using basic properties of ideals as described in [35, 4§3 Thm. 4], it is easy to show that $G \models_{\text{bool}} f \iff G \cup B \models f$, where $B = \{x_i(x_i - 1) : i = 1, \dots, n\}$. Furthermore, the equivalence $G \cup B \models f \iff G \cup B \vdash^+ f$ also holds when \mathbb{F} is not algebraically closed, because changing from \mathbb{F} to its algebraic closure $\bar{\mathbb{F}}$ will not have any effect on the models in $\{0, 1\}^n$. Finally, let us remark that the finiteness of $\{0, 1\}^n$ also implies that $G \cup B \vdash^+ f \iff G \cup B \vdash f$. This follows from Seidenberg's lemma [9, Lemma 8.13] and generalizes Theorem 1 of [34].

In contrast to a PC refutation $G \cup \{1 - yf\} \cup B \vdash 1$, where each polynomial in the proof is generated using the rules of PC, a refutation in the NS proof system is a set of

letter ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
 number ::= '0' | '1' | ... | '9'
 constant ::= (number)⁺
 variable ::= letter (letter | number)*
 power ::= variable ['^' constant]
 term ::= power ('*' power)^{*}
 monomial ::= constant | [constant '*'] term
 operator ::= '+' | '-'
 polynomial ::= ['-'] monomial (operator monomial)*
 given ::= (polynomial ';')^{*}
 rule ::= ('+' | '*') ':' polynomial ',' polynomial ',' polynomial ';' '
 proof ::= (rule ';')^{*}

Figure B.2: Syntax of given polynomials and proofs in PAC-format.

polynomials $Q = \{q_1, \dots, q_m\} \subseteq \mathbb{F}[X]$ such that

$$\sum_{i=0}^m q_i p_i = 1 \quad \text{for} \quad p_i \in G \cup \{1 - yf\} \cup B.$$

Although both systems are able to verify correctness of a refutation, we will use PC and not the NS proof system, because for arithmetic circuit verification we will rewrite some polynomials of $G \cup \{1 - yf\} \cup B$, and thus gain an optimized algebraic representation of the circuit, cf. Sect. B.4. In a correct NS refutation we would also need to express these rewritten polynomials as a linear combination of elements of $G \cup \{1 - yf\} \cup B$ and thus lose the optimized representation, which will most likely lead to an exponential blow-up of monomials in the NS proof [27]. In PC we can generate these optimized polynomials on-the-fly and then use these polynomials to show the correctness of the refutation.

B.3 Practical Algebraic Calculus

For practical proof checking we translate the abstract polynomial calculus (PC) into a concrete proof format, i.e., we only define a format based on PC, which is logically equivalent but more precise. In principle a proof in PC can be seen as a finite sequence of polynomials derived from given polynomials and previously inferred polynomials by applying either an addition or multiplication rule.

To ensure correctness of each rule it is of course necessary to know which rule was used, to check that it was applied correctly, and in particular which given or previously derived polynomials are involved. During proof generation these polynomials are usually known and thus we require that all of this information is part of a rule in our concrete *practical algebraic calculus* (PAC) proof format to simplify proof checking. The syntax of PAC is shown in Fig. B.2. White space is allowed everywhere except between letters

and digits in a constant or a variable. A proof rule contains four components

$$o : v, w, p;$$

The first component o denotes the operator which is either ‘+’ for addition or ‘*’ for multiplication. The next two components v, w specify the two (antecedent) polynomials used to derive p (conclusion). In the multiplication rule w plays the role of the polynomial q of the multiplication rule of PC, cf. Sect. B.2. A refutation in PAC is a proof, which contains a non-zero constant polynomial (typically just the constant “1”) as conclusion p of a rule.

As discussed above we do not need the radical rule for our purpose, even though it could be easily added. Further note that the format is independent of the domain of the models u , e.g., $u \in \{0, 1\}^n$ for gate-level circuit verification, to which the values of variables are restricted. If such a restriction is necessary, all elements of the corresponding set B (often also called field polynomials) have to be added to the given set of polynomials.

Although the definition of number together with the definition of polynomial only allows integer coefficients, this is not a severe restriction. Rational number coefficients can be simulated by multiplying involved polynomials with appropriate non-zero constants to eliminate denominators.

Example B.3. Consider again Ex. B.1. To test membership of $1 - c \in \sqrt{\langle G \rangle}$ we add $1 + y(c - 1)$ to the set of given polynomials G in order to apply Rabinowitsch’s trick and obtain a PAC refutation:

$$\begin{array}{lll} + : & -c+a+b-2a*b, & -b+1-a, & -c+1-2a*b; \\ * : & -b+1-a, & -2a, & 2a*b-2a+2a^2; \\ + : & -c+1-2a*b, & 2a*b-2a+2a^2, & -c+1-2a+2a^2; \\ * : & a^2-a, & -2, & -2a^2+2a; \\ + : & -c+1-2a+2a^2, & -2a^2+2a, & -c+1; \\ * : & -c+1, & y, & -c*y+y; \\ + : & -c*y+y, & 1+c*y-y, & 1; \end{array}$$

For proof validation we need to make sure that two properties hold. The *connection property* states that the components v, w are either given polynomials or conclusions of previously applied proof rules. For multiplication we only have to check this property for v , because w is an arbitrary polynomial. By the second property, called *inference property*, we verify the correctness of each rule, namely we simply calculate $v + w$ resp. $v * w$ and check that the obtained result matches p . In a correct PAC refutation we further need to verify that at least one p_i is a non-zero constant. The complete checking algorithm is shown in Alg. 5.

B.4 Circuit verification using Computer Algebra

Following [32, 89, 91, 93, 94, 103] we consider gate-level (integer) multipliers with $2n$ input bits $a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1} \in \{0, 1\}$ and $2n$ output bits $s_0, \dots, s_{2n-1} \in$

Algorithm 5: Proof Checking algorithm in PAC

Input : G sequence of given polynomials, $r_1 \cdots r_k$ sequence of PAC proof rules
Output : “incorrect”, “correct-proof”, or “correct-refutation”

```

1  $P_0 \leftarrow G$ ;
2 for  $i \leftarrow 1 \dots k$  do
3   let  $r_i = (o_i, v_i, w_i, p_i)$ ;
4   case  $o_i = +$  do
5     if  $v_i \in P_{i-1} \wedge w_i \in P_{i-1} \wedge p_i = v_i + w_i$  then  $P_i \leftarrow \text{append}(P_{i-1}, p_i)$ ;
6     else return “incorrect”;
7   case  $o_i = *$  do
8     if  $v_i \in P_{i-1} \wedge p_i = v_i * w_i$  then  $P_i \leftarrow \text{append}(P_{i-1}, p_i)$ ;
9     else return “incorrect”;
10 for  $i \leftarrow 1 \dots k$  do
11   if  $p_i$  is a non zero constant polynomial then return “correct refutation”;
12 return “correct proof”;

```

$\{0, 1\}$. Each internal gate (output) is represented by a further variable l_1, \dots, l_m . In this setting let $X = a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1}, l_1, \dots, l_m, s_0, \dots, s_{2n-1}$. Then a multiplier is correct iff for all possible inputs the following specification holds:

$$\sum_{i=0}^{2n-1} 2^i s_i = \left(\sum_{i=0}^{n-1} 2^i a_i \right) \left(\sum_{i=0}^{n-1} 2^i b_i \right) \quad (\text{B.1})$$

Using algebraic reasoning this can be verified by showing that the specification is contained in the ideal generated by the gate constraints. For each logical gate in the circuit a so-called *gate polynomial* $g \in \mathbb{Q}[X]$ representing the relation between the gate inputs and output is defined. Example B.1 defines these polynomials for a NOT and an XOR gate. Indicating that the circuit operates over Boolean variables, we add for each variable $x_i \in X$ the relation $x_i(x_i - 1)$ matching the definition of B in the last paragraph of Sect. B.2 to the gate polynomials G .

Although all variables are restricted to boolean values we use \mathbb{Q} as the base field. Using \mathbb{Q} connects the circuit specification (Eqn. (B.1)) to multiplication in \mathbb{Q} . The specification would be the same over \mathbb{Z} , but \mathbb{Z} is not a field, hence the underlying Gröbner basis theory would be more complex. Theoretically reasoning in the field \mathbb{Z}_2 is possible, but probably would be much more involved. A more precise comparison will be done in the future.

A term order is 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 $\sigma_1 < \sigma_2$ iff there exists i with $u_j = v_j$ for all $j < i$, and $u_i < v_i$. If the terms in the gate polynomials are ordered according to such a lexicographic variable ordering where the variable corresponding to the output of a gate is always bigger than the variables corresponding to inputs of the gate, then by Buchberger’s product criterion [35] the gate polynomials define a Gröbner basis for the ideal generated by

the gate polynomials. Thus the correctness of the circuit can be shown by reducing the specification by the gate polynomials using polynomial reduction (red_G) and checking if the result is zero. We generate and check proofs for this reduction, cf. Sect. B.5.

Directly reducing the specification without rewriting the Gröbner basis leads to an explosion of intermediate results [89]. In practice it is necessary to use rewriting techniques to simplify the Gröbner basis. In recent work [93] a reduction scheme was proposed which effectively (partially) reduces the Gröbner basis. These preprocessing steps [93] are also applied in [89], where we introduced a column-wise checking algorithm which cuts the circuit into $2n$ slices S_i with $0 \leq i < 2n$ such that each slice contains exactly one output bit s_i . In each slice the relation that the sum of the outgoing carries C_{i+1} and the output bit s_i is equal to the sum of the partial products $P_i = \sum_{k+l=i} a_k b_l$ and the incoming carries of the slice C_i has to hold. Thus we define for each slice S_i a corresponding specification $C_i = 2C_{i+1} + s_i - P_i$. Initially we set $C_{2n} = 0$ and recursively calculate C_i as the remainder of reducing $2C_{i+1} + s_i - P_i$ by the gate polynomials of the corresponding slice. In a correct multiplier $C_0 = 0$ has to hold. Hence each slice is verified recursively, thus the problem of circuit verification is divided into smaller more manageable sub-problems.

In [91] we further improved incremental checking by eliminating variables [17], local to full- and half-adders. Since these preprocessing and incremental algorithms are complex and error prone to implement but essential to achieve scalable verification, we also generate and check proofs for them.

B.5 Engineering

We take as input circuit an And-Inverter Graph (AIG) [70] in the common AIGER format [16]. The AIG is then verified using the computer algebra system Mathematica [102]. We also generate proofs in our PAC-format (c.f. Sect. B.3) which then are either passed on to the computer algebra system Singular [38] or to our own algebraic proof checker PACTRIM. The complete verification flow is depicted in Fig. B.3. Boxes with “.(suffix)” refer to the input AIG or generated files. The variable n defines the length of the two input bit-vectors of the multiplier.

The tool AIGMULTOPOLY [89, 91] is used for verification without generating proofs (**verify**). It takes an AIG as input and produces a file which can be passed on to either Mathematica or Singular, which then performs the actual ideal membership test. Different option settings can be selected to enable or disable the preprocessing and rewriting techniques discussed in Sect. B.4.

For proof generation (**verify+**) we use a second tool PROOFIT which takes the output file from AIGMULTOPOLY as well as the original AIG and returns a file which can be passed on to Mathematica. In Mathematica the proof (.pac) is calculated. In the tool AIGTOPOLY the original AIG is translated into a set of polynomials G without applying any preprocessing. Together with the set $B = \{x_i(x_i - 1) \mid x_i \in X\}$ these polynomials define the given set of polynomials $G \cup B$ of the PAC proof (.polys). This

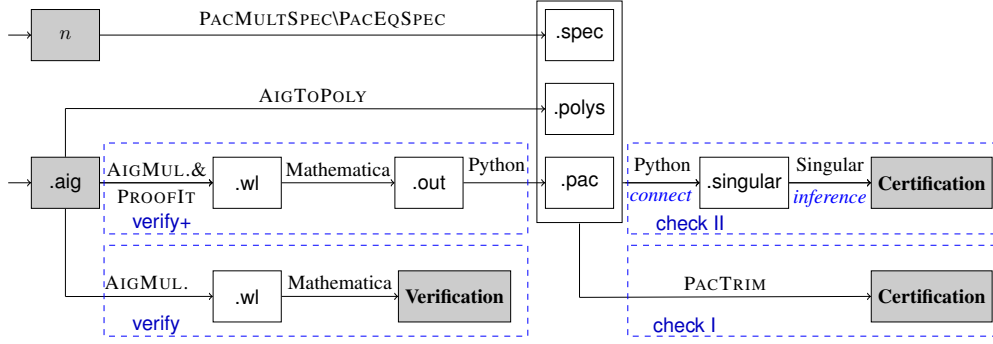


Figure B.3: Toolflow of verifying and proof checking circuits.

is a rather trivial task implemented in fewer than 130 lines of C code (half of them are just about command line option handling) using the AIGER [16] library for parsing.

In the same spirit PACMULTSPEC and PACEQSPEC have been implemented to produce the specifications we want to verify (.spec). In PACMULTSPEC we simply generate the multiplier specification as given in Sect. B.4, i.e., Eqn. (B.1) flattened. In PACEQSPEC we generate a similar specification for equivalence checking of two multipliers [91]. To gain a PAC refutation both types of specifications are produced in negated form using the Rabinowitsch trick and hence become part of the given set of polynomials.

Each polynomial of AIGMULTOPOLY which is derived during preprocessing needs to be checked if it is a logical consequence of the given set of polynomials. Hence for each preprocessed polynomial f the representation modulo the given set of polynomials $G \cup B = \{g_1, \dots, g_k\}$ is calculated in Mathematica using the built-in function “PolynomialReduce”. This command does not only allow computing the reduction $\text{red}_{G \cup B}(f) = r$, but it also returns cofactors h_1, \dots, h_k such that $f = h_1g_1 + \dots + h_kg_k + r$. If the preprocessing is done correctly, the derived polynomials f are contained in the ideal $\langle G \cup B \rangle$, thus $\text{red}_{G \cup B}(f) = 0$ and the above representation simplifies to $f = h_1g_1 + \dots + h_kg_k$. Knowing the cofactors h_i and the corresponding elements of $G \cup B$ we generate proof rules in PAC in the following way. First we generate a multiplication proof rule for each product $h_i g_i$.

$$* : g_1, h_1, h_1g_1; \quad \dots \quad * : g_k, h_k, h_kg_k;$$

In the listed rules the result p is always depicted simply as the product $h_i g_i$, but in the actual PAC proof p is written in expanded (flattened) form. These products are now simply added together as follows:

$$\begin{array}{ll} + : & h_1g_1, \quad h_2g_2, \quad h_1g_1 + h_2g_2; \\ + : & h_1g_1 + h_2g_2, \quad h_3g_3, \quad h_1g_1 + h_2g_2 + h_3g_3; \\ & \vdots \\ + : & h_1g_1 + \dots + h_{k-1}g_{k-1}, \quad h_kg_k, \quad f; \end{array}$$

In the experiments we also use a non-incremental verification approach where we do not use the incremental optimizations presented in Sect. B.4, hence we have to reduce the complete word-level specification of a multiplier by the (preprocessed) gate and field polynomials. Extracting a proof works in the same way as just described for the preprocessed polynomials.

Generating proofs for incremental verification is also similar, but instead of the word-level specification of the multiplier we have to use the *incremental specifications* $C_i = 2C_{i+1} + s_i - P_i$ of each slice, cf. Sect. B.4. The polynomials C_i describing the incoming carries of a slice can be derived by calculating $\text{red}_{G \cup B}(2C_{i+1} + s_i - P_i) = C_i$. Since verification can be assumed to succeed, we have $C_{2n} = 0$ and $C_0 = 0$. As described in the last bullet on fundamental facts in Sect. B.2 we are able to obtain cofactors h_1, \dots, h_k such that $2C_{i+1} + s_i - P_i - C_i = h_1g_1 + \dots + h_kg_k$ and consequently a translation into the PAC-format to derive the left-hand side of the equation.

To derive the word-level specification of a multiplier from the incremental specifications we first multiply for each slice S_i its incremental specification $C_i = 2C_{i+1} + s_i - P_i$ by the constant 2^i .

$$\begin{array}{lll} * : & 2C_1 + s_0 - P_0, & 1, \quad 2C_1 + s_0 - P_0; \\ * : & 2C_2 + s_1 - P_1 - C_1, & 2, \quad 4C_2 + 2s_1 - 2P_1 - 2C_1; \\ & \vdots & \\ * : & s_{2n-1} - P_{2n-1} - C_{2n-1}, & 2^{2n-1}, \quad 2^{2n-1}s_{2n-1} - 2^{2n-1}P_{2n-1} - 2^{2n-1}C_{2n-1}; \end{array}$$

Subsequent accumulation of the polynomials above using PAC addition rules cancels the terms C_i and $\sum_{i=0}^{2n-1} 2^i s_i - \sum_{i=0}^{2n-1} 2^i P_i$ remains. It holds that the sum of partial products can be reordered to $\sum_{i=0}^{2n-1} 2^i P_i = (\sum_{i=0}^{n-1} 2^i a_i)(\sum_{i=0}^{n-1} 2^i b_i)$ [89] and thus we are able to deduce the word-level specification of multipliers.

In both approaches the incremental as well as the non-incremental one, we multiply the word-level specification of the multiplier by the additional variable y and add it to the given polynomial $1 - y * \text{spec} \in G \cup B$ to derive 1 and thus obtain a correct PAC refutation.

As Fig. B.3 shows we have two different flows for checking PAC proofs independently from Mathematica, which was used for verification. The first one uses Python scripts to validate the *connection* property of each rule and whether the proof actually defines a refutation. With Singular we check the *inference* property of each proof line, which in essence uses Singular as a calculator for adding and multiplying polynomials.

We also provide a new dedicated proof checker called PACTRIM implemented from scratch in C. It has similar features as DRAT-TRIM, which is the standard proof checker in the SAT community for clausal proofs (and is used in the SAT Competition – see also [46, 51]). Our new PACTRIM checker contains a parser for PAC proofs and checks the *connection* property using hash tables and the *inference* property using a dedicated stand-alone implementation of polynomial arithmetic over arbitrary precision integers represented as strings.

While the first approach is rather general and easy to adapt it is, as the experiments confirm, less robust (due to for instance the limit on variables in Singular) and more

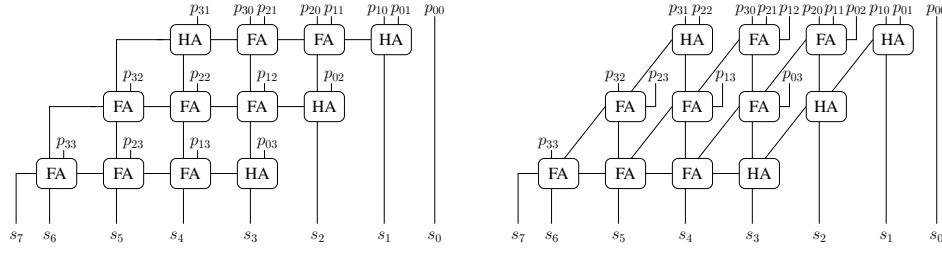


Figure B.4: Architecture of “btor” (left) and “sparrc” (right), where $p_{ij} = a_i b_j$ [91].

importantly far less efficient than our dedicated checker. The latter also allows to produce proof cores (of both original polynomials and proof lines), and is also much closer to being certifiable.

B.6 Experiments

In our experiments we generate and validate PAC proofs for the (integer) multiplier benchmarks used in [89,91]. The “btor”-benchmarks are generated by Boolector [83] and the “sparrc”-multipliers are part of the bigger AOKI benchmark set [53], containing several multiplier architectures. In both multiplier architectures the partial products are generated as products of two input bits which are then accumulated by full- and half-adders, as shown in Fig. B.4 for input size $n = 4$. In “btor”-multipliers the full- and half-adders are accumulated in a grid-like structure, thus they are considered as array multipliers, whereas in “sparrc”-multipliers full- and half-adders are accumulated diagonally.

In all our experiments we use a standard Ubuntu 16.04 Desktop machine with Intel i7-2600 3.40GHz CPU and 16 GB of main memory. The (wall-clock) time limit is 90 000 seconds and the main memory usage is limited to 7GB. The time in our experiments is measured in seconds (wall-clock time). We mark unfinished experiments by TO (reached time limit), MO (reached memory limit) or by EE, when an error state is reached. An error state is reached by Singular, because it has a limit of 32767 on the number of ring variables. All experimental data, benchmarks and source code is available at <http://fmv.jku.at/pac>.

In Table B.1 we separately list the time taken for verification, the generation as well as checking of PAC-proofs for “btor” and “sparrc” multipliers of different input bit-width n . The third column lists configurations of AIGMULTOPOLY. The default configuration uses incremental column-wise slicing of [89], c.f. Sect. B.4, both with (inc-add) and without (inc) our new optimization of eliminating local variables in full- and half-adders [91]. In the third configuration (noninc) the whole word-level specification is reduced without any slicing of the multiplier.

The time needed for verification, proof generation and proof checking is listed in the following columns. The corresponding execution paths are marked in Fig. B.3 by dashed rectangles. The column **verify** shows the time Mathematica needs to verify

n	mult	option	verify	verify+	chk I	con	inf	chk II	length	core	size	core	deg
4	btor	inc	0	1	0	0	0	0	646	68%	3551	72%	6
4	btor	inc-add	0	1	0	0	0	0	594	62%	4001	63%	5
4	btor	noninc	0	1	0	0	0	0	638	68%	3862	74%	6
8	btor	inc	1	4	0	0	0	0	3350	65%	21169	70%	6
8	btor	inc-add	0	3	0	0	0	0	2914	62%	21915	64%	5
8	btor	noninc	1	5	0	0	0	0	3334	65%	28227	78%	6
16	btor	inc	4	70	0	1	3	4	14998	64%	106853	72%	6
16	btor	inc-add	1	37	0	1	3	4	12738	61%	104351	66%	5
16	btor	noninc	4	78	0	1	9	10	14966	64%	231643	87%	6
32	btor	inc	44	1631	1	26	57	83	63254	64%	533773	76%	6
32	btor	inc-add	7	801	1	18	49	67	53122	61%	487911	69%	5
32	btor	noninc	65	1811	5	29	522	551	63190	64%	2594059	95%	6
64	btor	inc	622	49638	4	586	4539	5125	259606	63%	2839901	81%	6
64	btor	inc-add	121	22378	4	414	4236	4650	216834	61%	2387831	74%	5
64	btor	noninc	MO	MO	-	-	-	-	-	-	-	-	-
4	sparre	inc	0	1	0	0	0	0	753	64%	4943	68%	6
4	sparre	inc-add	0	1	0	0	0	0	764	65%	8156	66%	8
4	sparre	noninc	0	1	0	0	0	0	745	65%	5252	71%	6
8	sparre	inc	1	8	0	0	0	0	3917	62%	30494	69%	6
8	sparre	inc-add	0	7	0	0	1	1	3964	63%	59330	63%	8
8	sparre	noninc	1	33	0	0	0	1	3901	63%	37477	75%	6
16	sparre	inc	8	134	0	2	6	7	17445	62%	152698	71%	6
16	sparre	inc-add	1	112	0	2	18	20	17804	63%	317874	62%	8
16	sparre	noninc	11	2696	0	2	15	17	17413	62%	276885	84%	6
32	sparre	inc	104	3582	1	43	132	175	73301	62%	735218	74%	6
32	sparre	inc-add	8	2611	2	55	402	457	75244	63%	1492082	63%	8
32	sparre	noninc	351	TO	-	-	-	-	-	-	-	-	-
64	sparre	inc	1575	TO	-	-	-	-	-	-	-	-	-
64	sparre	inc-add	133	80906	12	1307	EE	EE	309164	62%	6727026	65%	8
64	sparre	noninc	MO	-	-	-	-	-	-	-	-	-	-

Table B.1: Wordlevel proof checking.

n	mult	verify	verify+	chk I	con	inf	chk II	length	core	size	core	deg
4	btor-btor	1	1	0	0	1	1	1170	59%	7952	61%	5
8	btor-btor	1	6	0	0	1	1	5794	59%	43902	63%	5
16	btor-btor	2	75	1	5	10	14	25410	59%	210666	65%	5
32	btor-btor	27	1632	3	87	189	277	106114	59%	995330	69%	5
64	btor-btor	502	45155	15	1625	EE	EE	433410	59%	4942642	74%	5
4	btor-sparrc	1	2	0	1	1	2	1340	61%	12107	64%	8
8	btor-sparrc	1	9	1	1	2	3	6844	61%	81317	63%	8
16	btor-sparrc	3	148	1	7	42	48	30476	61%	424189	63%	8
32	btor-sparrc	28	3456	7	163	848	1011	128236	60%	1999501	64%	8
4	sparrc-sparrc	1	2	0	0	0	1	1510	62%	16270	65%	8
8	sparrc-sparrc	1	12	1	1	5	6	7894	62%	118820	63%	8
16	sparrc-sparrc	2	223	2	9	73	82	35542	61%	638248	62%	8
32	sparrc-sparrc	29	5363	11	308	1591	1899	150358	61%	3006256	63%	8

Table B.2: Equivalence proof checking.

the multiplier, column **verify+** shows the time needed to generate the proof including the time of **verify** and in column **chk I** we measure the time our own proof checker PACTRIM needs to validate the proof. The time Python needs to verify the *connection* property is listed in column **con** and the time Singular needs to verify the *inference* property is listed in column **inf**. The column **chk II** is the total time needed to verify the proof with Python and Singular. We did not include the time the tools AIGTOPOLY, PACMULTSPEC and PACEQSPEC need, because in the worst-case it only takes a second for 64-bit multipliers.

Inspired by [82] we also compute and include the number of polynomials in a proof (length), the total number of monomials of the derived polynomials (size), counted with repetition, and the maximum total degree of any monomial (deg). Usually not all given polynomials in the data set $G \cup \{1 - yf\} \cup B$ are needed to derive a correct refutation, especially only a small subset of B is used. Thus next to the length and size columns we list the percentage of polynomials and monomials which are actually necessary to derive a PAC refutation (core) w.r.t. the number of original and derived polynomials.

In general it can be seen that “sparrc”-multipliers need more time and space for verification, certification and proof checking than “btor”-multipliers. By far most of the time is needed for generating the proofs. For more scalable proof generation it is clear that computer algebra systems would need to be adapted to support proof generation on-the-fly or even application specific algebraic reasoning engines have to be implemented. Checking the proof with PACTRIM takes a fraction of the time needed for verification, at most 12 seconds, even for 64-bit multipliers. Proof checking using an independent computer algebra system takes much longer – for 64-bit multipliers more than 4000 seconds.

In further experiments shown in Table B.2 we construct proofs for the commutativity property of multipliers, i.e., we want to prove for a certain multiplier architecture that

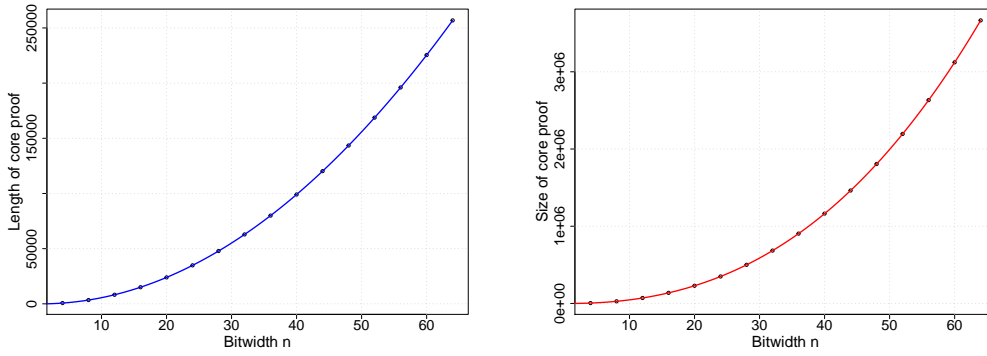


Figure B.5: Length and size of btor-btor commutativity check.

$A*B = B*A$ holds. Among other things it was shown in the work of [7] that polynomial sized resolution proofs for the commutativity property of array and diagonal multipliers exist. Motivated by this result we generate proofs for these two multiplier architectures, where “btor”-multipliers play the role of array multipliers and “sparrc”-multipliers are considered as diagonal multipliers. We generate the commutativity miterers by checking the equivalence of a multiplier and the same multiplier with input bit-vectors swapped (btor-btor, sparrc-sparrc). Furthermore we derive proofs for checking the equivalence of the two architectures “btor” vs. “sparrc” (btor-sparrc). The columns in Table B.2 follow the same structure as in Table B.1. In all commutativity and equivalence checking experiments we used the configuration “inc-add”, which uses our incremental column-wise slicing of [89] with the optimization of eliminating local variables in full- and half-adders. We did not include commutativity or equivalence checking experiments containing “sparrc” multipliers with bit-width $n = 64$, because we reached an error state (EE) in the experiments of Table B.1.

In Fig. B.5 data points depicting core size (left plot) and core length (right plot) of the “btor-btor”-commutativity proofs are shown for various input bit-widths n . The additional polynomial curves are fitted to the data points (using linear regression with R). For the proof length we used a parameterized model of a quadratic polynomial. The proof size required a cubic polynomial. In both cases the match is perfect, with absolute values of residuals less than $9 * 10^{-10}$. This empirically suggested quadratic complexity of algebraic proofs compares favorably to the $\mathcal{O}(n^7 \log n)$ upper bound for resolution proofs given in Thm. 2 of [7].

Comparing the meta data of the “btor-btor” and “sparrc-sparrc”-benchmarks the proof lengths of “sparrc-sparrc”-benchmarks are of the same magnitude as the proof lengths of “btor-btor”-benchmarks. The proof sizes of “sparrc-sparrc” are around three times as big as the proof sizes of “btor-btor” with nearly same percentages for the cores. Hence both measurements of “sparrc-sparrc”-benchmarks can also be depicted by quadratic and cubic curves.

B.7 Conclusion

This paper applies proof checking to algebraic reasoning, not only in theory, but also in practice, in order to validate verification techniques based on computer algebra. We show how the abstract polynomial calculus [34] can be instantiated to yield a practical proof format (PAC). Proofs in this format can be obtained as by-product of verifying multiplier circuits using state-of-the-art techniques and can be checked with our new proof checker tool PACTRIM in a fraction of the time needed for verification. Our experiments produce small polynomial proofs which certify the correctness of certain multipliers. The theoretical analysis in [7] gives much larger polynomial upper bounds (for clausal resolution proofs).

To explore the connection between PAC and clausal proof systems, such as RUP and DRAT [47], is an interesting subject for future work, as well as embedding PAC into more general systems, such as Isabelle [84].

We want to thank Thomas Sturm for pointing out the Rabinowitsch trick to the second author and Jakob Nordström for discussions on the polynomial calculus and Nullstellensatz proof systems.



Paper C

Verifying Large Multipliers by Combining SAT and Computer Algebra

Published In Proceedings of the 19th International Conference on Formal Methods in Computer Aided Design (FMCAD 2019), pages 28–36, San Jose, CA, USA, 2019.

Authors Daniela Kaufmann, Armin Biere and Manuel Kauers.

Acknowledgement This work is supported by the Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), P31571-N32, SFB F5004, LIT AI Lab funded by the state of Upper Austria.

Abstract We combine SAT and computer algebra to substantially improve the most effective approach for automatically verifying integer multipliers. In our approach complex final stage adders are detected and replaced by simple adders. These simplified multipliers are verified by computer algebra techniques and correctness of the replacement step by SAT solvers. Our new dedicated reduction engine relies on a Gröbner basis theory for coefficient rings which in contrast to previous work no longer are required to be fields. Modular reasoning allows us to verify not only large unsigned and signed multipliers much more efficiently but also truncated multipliers. We are further able to generate and check proofs an order of magnitude faster than in our previous work, relative to verification time, while other competing approaches do not provide certificates.

C.1 Introduction

Automated formal verification of arithmetic circuits, most prominently multiplier circuits, remains an important problem, which in practice still requires substantial manual effort. Currently the most effective approach for automatically verifying integer multi-

pliers is based on polynomial reasoning using computer algebra techniques [31, 61, 79, 80, 106].

However, parts of multipliers, i.e., final stage adders, are a real challenge for the computer algebraic approach. In certain adder designs carries are computed by complex tree structures, leading to an explosion of intermediate results. Contrarily SAT solvers can easily verify the equivalence of adders. Therefore we replace complex final stage adders by simpler adders and verify the correctness of the replacement using SAT solvers. The simplified multiplier is verified using computer algebra.

Our new dedicated reduction engine makes use of the structure of the polynomial representation of circuits and is more capable in multiplier verification than computer algebra systems [38, 102] used in our previous work. Additionally it efficiently produces certificates, which validate the correctness of the verification. In previous work we used \mathbb{Q} as coefficient domain. Our experiments show that it is beneficial to use more general rings to support modular arithmetic [100]. We summarize the theory and provide arguments for the correctness of the algebraic approach over more general rings. As a consequence we are able to verify not only unsigned and signed multipliers more efficiently but also truncated multipliers.

Recently significant progress has been made in fully automated verifying integer multipliers using computer algebra. The authors of [79, 80] presented a method allowing local cancellation of vanishing monomials in converging gate cones. This approach is empirically much more successful than previous work in verifying a large variety of multiplier architectures but its formal exposition has room for improvement. The technique of [31, 106] eliminates redundant polynomials by identifying and rewriting half- and full-adders in the circuit. This approach is able to verifying large clean multiplier circuits, but fails on complex multiplier architectures. None of these methods produces certificates. Contrarily theorem provers in combination with SAT are able to certify industrial multipliers [54], however this approach is not fully automated.

C.2 Specifying Multiplier Circuits

A circuit implements a logical function and the specification of a circuit is a desired relation between the inputs and the outputs of a circuit. We say that 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 and hence deriving correctness.

We consider acyclic gate-level circuits C with inputs a_0, \dots, a_{k-1} , a number of internal logical gates $g_1, \dots, g_l \in \{0, 1\}$, and outputs s_0, \dots, s_{m-1} in $\{0, 1\}$. Thus we fix X to denote the variables $a_0, \dots, a_{k-1}, g_1, \dots, g_l, s_0, \dots, s_{m-1}$. In the algebraic verification approach every input and output of a gate in the circuit is labeled by a variable and for each gate there is a polynomial describing the relation of the input and output variables of the gate. Correctness of the circuit is shown by proving that the specification, encoded as a polynomial \mathcal{L} , is implied by the polynomial relations of the gates.

Part of the specification is the ring to which the polynomial \mathcal{L} belongs. The circuit polynomials are also considered as elements of this ring. In our previous work [61] we chose the ring $\mathbb{Q}[X]$. We will now generalize the formulation of circuit verification using computer algebra to other polynomial rings. Our experiments show that by doing so we gain an enormous speed-up in the computation time. Although not formally introducing the theory, related work also relies on more general rings than $\mathbb{Q}[X]$. Furthermore by generalizing the theory to arbitrary polynomial rings we are able to verify different types of multipliers, which we now present. We fix a polynomial ring $R[X]$ and state the corresponding specification of the multipliers as an element of $R[X]$.

Definition C.1. Let $\varphi: X \rightarrow \{0, 1\} \subseteq R$ denote an *assignment* of all variables X . We extend φ to an evaluation of polynomials in the natural way, i.e., $\varphi: R[X] \rightarrow R$.

To capture multiplication of unsigned integers we consider circuits with two unsigned binary input bit-vectors $A = a_{n-1}, \dots, a_0$ and $B = b_{n-1}, \dots, b_0 \in \{0, 1\}^n$ and an output bit-vector $S = s_{2n-1}, \dots, s_0 \in \{0, 1\}^{2n}$, calculating $A \cdot B = S$.

In previous work [89] we verified unsigned integer multipliers, which most naturally are specified over \mathbb{Z} .

Definition C.2. The word-level specification \mathcal{U}_n of n -bit unsigned integer multipliers in the ring $\mathbb{Z}[X]$ is given as

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

A common way to represent signed integers is using two's complement. The value w of a bit-vector $K = k_{n-1}, \dots, k_0$ of length n in two's complement is given as

$$w = -2^{n-1} k_{n-1} + \sum_{i=0}^{n-2} 2^i k_i.$$

Thus in specifying signed multipliers we interpret A , B and S as *signed* bit-vectors in two's complement representation.

Definition C.3. The specification \mathcal{S}_n of n -bit signed integer multipliers in the ring $\mathbb{Z}[X]$ is given as

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

Our experiments will show that it is beneficial to use modular arithmetic to reduce the size of intermediate verification results. For now we still keep \mathbb{Z} as coefficient domain for the specification of unsigned and signed multipliers. Theorem C.23 and C.24, in

addition with Lemma C.21 and Lemma C.22, will then allow us to use $\mathbb{Z}_{2^{2n}}[X]$ too, because for any assignment φ the range of the specification does not exceed $\pm 2^{2n}$.

In contrast to the multipliers presented so far a truncated multiplier returns only n output bits for input bit-width n . In the result of multiplying two integers the n most significant bits are simply discarded. Thus a truncated multiplier calculates $A \cdot B = S \bmod 2^n$. We define the specification of *truncated multipliers* to be an element of the ring $\mathbb{Z}_{2^n}[X]$, because \mathbb{Z}_{2^n} is the ring whose multiplication we wish to describe.

Definition C.4. The specification \mathcal{T}_n of n -bit truncated multipliers in the ring $\mathbb{Z}_{2^n}[X]$ is given as

$$\mathcal{T}_n = \sum_{i=0}^{n-1} 2^i s_i - \sum_{i=0}^{n-1} \sum_{j=0}^{n-1-i} 2^{i+j} a_i b_j$$

Our theory is not limited to these multiplication instances. A further example would be Galois field multipliers, where the specification is an element of $\mathbb{Z}_2[X]/\langle p \rangle$, for a given irreducible polynomial p [78, 105]. Other possible (though perhaps useless) choices are rings of the form $\mathbb{Z}_m[X_1, \dots, X_k]/I$ for some given $m \in \mathbb{N}$ and some given ideal I .

C.3 Algebra

In our previous work [61] we outlined the underlying theory of circuit verification using computer algebra for polynomial rings over fields. In this section we generalize the theory to be applicable in more general polynomial rings. To this end let R be a commutative ring with unity and let $R[X]$ with $X = \{x_1, \dots, x_r\}$ be the polynomial ring over R . By R^\times we denote the set of multiplicatively invertible elements of R .

Definition C.5. A *term* $\tau = x_1^{d_1} \cdots x_r^{d_r}$ is a product of powers of variables for certain $d_1, \dots, d_r \in \mathbb{N}$. We denote the set of terms by $[X]$. A *monomial* is a multiple of a term $c\tau$, with $c \in R$ and a *polynomial* p 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 it holds that $1 \leq \tau$ and $\sigma_1 \leq \sigma_2 \Rightarrow \tau\sigma_1 \leq \tau\sigma_2$. Such an order is called a *lexicographic term order* if for all terms $\sigma_1 = x_1^{d_1} \cdots x_r^{d_r}$, $\sigma_2 = x_1^{e_1} \cdots x_r^{e_r}$ we have $\sigma_1 < \sigma_2$ iff there exists an index i with $d_j = e_j$ for all $j < i$, and $d_i < e_i$. For a polynomial $p = c\tau + \dots$ the largest term τ (w.r.t. \leq) is called the *leading term* $\text{lt}(p) = \tau$. Furthermore $\text{lc}(p) = c$ is called the *leading coefficient* and $\text{lm}(p) = c\tau$ is called the *leading monomial* of p . Then we call $p - c\tau$ the *tail* of p .

In the circuit the semantics of the logic gates imply polynomial relations among the variables, such as:

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

The polynomial equations are chosen in such a way that the possible solutions with $u, v, w \in \{0, 1\}$ of the polynomials are the solutions of the gate constraints and vice versa. As the left side is always zero, we take the freedom to write f instead of $0 = f$. Note that the coefficients 1, -1 and 2 are elements of the ring R , where 1 represents the unity of R , -1 represents its additive inverse, and $2 = 1 + 1$. On the set of terms we fix a lexicographic term order, called *reverse topological term order*, such that the output variable of a gate is always greater than the variables attached to the input edges of that gate.

By $G(C) \subseteq R[X]$ we denote the set of *circuit polynomials* which contains for each gate of C the corresponding polynomial of (C.2). To encode that each variable $x \in X$ represents a boolean value, we further have *boolean value constraints* $x(1 - x) = 0$. Let $B(Y) = \{y(1 - y) \mid y \in Y\} \subseteq R[X]$ for $Y \subseteq X$, be the set of boolean value constraints for Y . By \mathcal{L} we denote the polynomial in $R[X]$ which models the *specified* relation between the input and outputs of the circuit.

Definition C.6. A nonempty subset $I \subseteq R[X]$ is called an *ideal* if $\forall p, q \in I : p + q \in I$ and $\forall p \in R[X] \forall q \in I : pq \in I$. A set $P = \{p_1, \dots, p_s\} \subseteq R[X]$ is called a *basis* of I if $I = \{p_1q_1 + \dots + p_sq_s \mid q_1, \dots, q_s \in R[X]\}$. We say I is generated by P and write $I = \langle P \rangle$. The sum of two ideals I and J is defined as $I + J = \{p + q \mid p \in I, q \in J\}$.

Note, if $I = \langle P \rangle$ and $J = \langle Q \rangle$ are ideals generated by $P, Q \subseteq R[X]$, then $I + J = \langle P \rangle + \langle Q \rangle = \langle P \cup Q \rangle$.

We show that the question whether \mathcal{L} is implied by the gate polynomials of C and the boolean value constraints can be answered by a so-called ideal membership test:

“Given a polynomial $q \in R[X]$ and a (finite) set of polynomials $P \subseteq R[X]$, decide whether $q \in \langle P \rangle$.”

Definition C.7. Let $P \subseteq R[X]$. If for a certain term order, all leading terms of P only consist of a single variable with exponent 1 and are unique and further $\text{lc}(p) \in R^\times$ 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 C.8. The set $P = \{-x + 2y, y - z\} \subseteq \mathbb{Z}[x, y, z]$ 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. Note, in our application the leading coefficients of the polynomials in $G(C)$ are only ± 1 . However we prefer the more general statement of Def. C.7, allowing that Thm. C.11 and Thm. C.15 also work for more general settings.

Definition C.9. Let $P \subseteq R[X]$ be a finite set of polynomials with UMLT. A polynomial $q \in R[X]$ can be *deduced* from P if $q \in \langle P \rangle + \langle B_0(P) \rangle$. In this case we write $P \vdash_R q$.

Definition C.10. For a given set $P \subseteq R[X]$, a *model* is an assignment φ such that for all $p \in P$ we have $\varphi(p) = 0$. For a set $P \subseteq R[X]$ and a polynomial $q \in R[X]$, we write $P \models_R q$ if every model for P is also a model for $\{q\}$, i.e., $P \models_R q \iff \forall \varphi : \forall p \in P : \varphi(p) = 0 \Rightarrow \varphi(q) = 0$.

Note, that for the purpose of this paper, these notions of syntactic “deduction” and semantic “models” are restricted to our application where variables take only boolean values.

Theorem C.11 (Soundness). *Let $P \subseteq R[X]$ be a finite set of polynomials with UMLT and $q \in R[X]$, then*

$$P \vdash_R q \Rightarrow P \models_R q.$$

Proof. If $P \vdash_R q$ then $q \in \langle P \rangle + \langle B_0(P) \rangle$ by definition. This means there are $u_1, \dots, u_m \in R[X]$ and $v_1, \dots, v_r \in R[X]$ with $q = u_1 p_1 + \dots + u_m p_m + v_1 b_1 + \dots + v_r b_r$, where $p_i \in P$ and $b_i = x_i(x_i - 1) \in B_0(P) \subseteq B(X)$ for $i = 1 \dots r$. Any assignment φ vanishes on $B(X)$, i.e., $\varphi(b_i) = 0$. If φ is also a model of P then $\varphi(p_i) = 0$ too and as a consequence $\varphi(q) = 0$. Therefore $P \models_R q$, as claimed. \square

Completeness is not obvious. Consider for instance that $\{2x\} \models_{\mathbb{Z}} x$ but $x \notin \langle 2x \rangle$ in $\mathbb{Z}[X]$. Requiring P to have UMLT turns out to be essential (which $\{2x\}$ does not have in $\mathbb{Z}[X]$, because $2 \notin \mathbb{Z}^\times$).

Lemma C.12. *If $P \models_R p$ and $P \models_R q$ then $P \models_R q \pm p$.*

Lemma C.13. *Let $P \subseteq R[X]$ be a finite set of polynomials with UMLT. Then for all $q \in R[X]$ there exists $p \in \langle P \rangle + \langle B_0(P) \rangle$ and $r \in R[X_0(P)]$ with $q = p + r$, such that the monomials in r have only exponents 1.*

Proof. Since P has UMLT, we can replace every occurrence of a leading variable of P in q by the corresponding tail. This process has to terminate because the tail of a polynomial contains only smaller variables and the number of variables in P is finite. Thus at some point only variables in $X_0(P)$ are left which do not occur as leading terms. If these variables occur with exponent larger than one we can use $B_0(P)$ to reduce their exponent to 1, which yields r . All reduction steps to obtain r can be captured by adding polynomials $f \cdot g$ with $f \in R[X]$ and $g \in P \cup B_0(P)$. Their sum gives p . \square

Example C.14. Let $P \subseteq \mathbb{Z}[x, y, z]$ be as in Ex. C.8 and assume $q = 2x^2 + xy + z^2 \in \mathbb{Z}[x, y, z]$. Consequently

$$\begin{aligned} p &= (-2x-5y)(-x+2y) + (10y+10z)(y-z) - 11(-z^2+z) \\ &= 2x^2 + xy + z^2 - 11z \in \langle P \rangle + \langle B_0(P) \rangle \text{ and} \end{aligned}$$

$$r = 11z \in \mathbb{Z}[X_0(P)].$$

Theorem C.15 (Completeness). *Let $P \subseteq R[X]$ be a finite set of polynomials with UMLT. Then for every $q \in R[X]$ we have*

$$P \models_R q \Rightarrow P \vdash_R q.$$

Proof. Suppose we have $P \models_R q$. Then our goal is to show $q \in \langle P \rangle + \langle B_0(P) \rangle$. First, by applying Lemma C.13, we obtain $p \in \langle P \rangle + \langle B_0(P) \rangle$ and $r \in R[X_0(P)]$ with $q = p + r$. Thus $P \vdash_R p$ by definition. Using Thm. C.11 we derive $P \models_R p$ and accordingly $P \models_R q - p = r$ by Lemma C.12. Now assume $r \neq 0$ and let m be a monomial of r which contains the smallest number of variables. Consider the assignment φ that maps $x \in X_0(P)$ to 1 if it appears in m and to 0 otherwise. Therefore $\varphi(r) \neq 0$ since exponents of variables in r are all one. This assignment on $X_0(P)$ admits a unique extension to X which vanishes on P (e.g., if $-x + t \in P$ with leading monomial $-x$, then choose $\varphi(x) = \varphi(t)$). This contradicts $P \models_R r$. Thus $r = 0$ and $q = p + r \in \langle P \rangle + \langle B_0(P) \rangle$. \square

It is easy to see that for an acyclic circuit C the set $G(C)$ has UMLT for the fixed reverse topological term order. As a consequence Theorem C.11 and C.15 can be applied and show that deciding the correctness of circuits can be reduced to deciding ideal membership problems for $R[X]$.

Definition C.16. $I(C) = \{f \in R[X] : G(C) \models_R f\}$.

It also easily follows that $I(C)$ is an ideal and contains all the relations that hold among the values at the different signals (gates and inputs) of the circuit. Thus we are particularly interested whether the specification polynomial \mathcal{L} is in $I(C)$.

Definition C.17. A circuit C fulfills \mathcal{L} iff $\mathcal{L} \in I(C)$.

Definition C.18. Write $B_0(C) = B_0(G(C))$ for an acyclic circuit C and define $J(C) = \langle G(C) \cup B_0(C) \rangle$ in $R[X]$.

Note that $J(C)$ is generated by the gate polynomials $G(C)$ and the boolean value constraints on the variables $X_0(G(C))$ not occurring as leading term in $G(C)$. Further, by definition, $q \in J(C)$ iff $G(C) \vdash_R q$. Thus $J(C)$ contains exactly those polynomial constraints “deducible” from the circuit.

Corollary C.19. For all acyclic circuits C , it holds $I(C) = J(C)$.

Proof. By the choice of term order, $G(C)$ satisfies the necessary conditions of Thm. C.11 and Thm. C.15 and applying them allows to conclude $q \in I(C) \Leftrightarrow q \in J(C)$. \square

Corollary C.20. A circuit C fulfills \mathcal{L} iff $\mathcal{L} \in J(C)$.

In order to improve efficiency through modular reasoning (replacing \mathbb{Z} by $\mathbb{Z}_{2^{2n}}$) we show that the specifications for unsigned and signed multipliers remain correct for $\mathbb{Z}_{2^{2n}}$ too.

Lemma C.21. For all assignments $\varphi: X \rightarrow \{0, 1\}$ it holds that $\varphi(\mathcal{U}_n) \in [-2^{2n} + 1, 2^{2n} - 1]$ in \mathbb{Z} .

Proof. The maximum of $\varphi(\mathcal{U}_n)$ is reached for the assignment φ_{\max} with $\varphi_{\max}(s) = 1$ for all $s \in S$ and $\varphi_{\max}(x) = 0$ for $x \in A \cup B$. Consequently

$$\varphi_{\max}(\mathcal{U}_n) = \sum_{i=0}^{2n-1} 2^i = 2^{2n} - 1 < 2^{2n}.$$

The minimum of $\varphi(\mathcal{U}_n)$ is reached for the assignment φ_{\min} with $\varphi_{\min}(s) = 0$ for all $s \in S$ and $\varphi_{\min}(x) = 1$ for $x \in A \cup B$. It follows (assuming of course $n > 0$) that

$$\varphi_{\min}(\mathcal{U}_n) = -(2^n - 1)^2 = -2^{2n} + \underbrace{2^{n+1}}_{>2} - 1 > -2^{2n}. \quad \square$$

Lemma C.22. *For all assignments $\varphi: X \rightarrow \{0, 1\}$ it holds that $\varphi(\mathcal{S}_n) \in [-2^{2n} + 1, 2^{2n} - 1]$ in \mathbb{Z} .*

Proof. The maximum of $\varphi(\mathcal{S}_n)$ is reached for the assignment φ_{\max} with $\varphi_{\max}(s_i) = 1$ for all $0 \leq i \leq 2n - 2$ and $\varphi_{\max}(s_{2n-1}) = 0$ and $\varphi_{\max}(a_j) = 1$ for all $0 \leq j \leq n - 2$ and $\varphi_{\max}(a_{n-1}) = 0$ and $\varphi_{\max}(b_j) = 0$ for all $0 \leq j \leq n - 2$ and $\varphi_{\max}(b_{n-1}) = 1$. Then

$$\varphi_{\max}(\mathcal{S}_n) = 2^{2n-1} - 1 + 2^{n-1}(2^{n-1} - 1).$$

By transforming the inequality we gain the desired result.

$$\begin{aligned} 2^{2n-1} + 2^{2n-2} - 2^{n-1} - 1 &< 2^{2n} \\ 2^{2n-2}(2 + 1 - 4) - 2^{n-1} - 1 &< 0 \end{aligned}$$

The minimum of $\varphi(\mathcal{S}_n)$ is reached for the assignment φ_{\min} with $\varphi_{\min}(s_i) = 0$ for all $0 \leq i \leq 2n - 2$ and $\varphi_{\min}(s_{2n-1}) = 1$ and $\varphi_{\min}(a_j) = \varphi_{\min}(b_j) = 0$ for all $0 \leq j \leq n - 2$ and $\varphi_{\min}(a_{n-1}) = \varphi_{\min}(b_{n-1}) = 1$. It follows that

$$\begin{aligned} \varphi_{\min}(\mathcal{S}_n) &= -2^{2n-1} - (-2^{n-1})^2 \\ &= -3 \cdot 2^{2n-2} > -4 \cdot 2^{2n-2} = -2^{2n}. \end{aligned}$$

□

Theorem C.23. $G(C) \models_{\mathbb{Z}} \mathcal{U}_n$ iff $G(C) \models_{\mathbb{Z}_{2^{2n}}} \mathcal{U}_n$.

Theorem C.24. $G(C) \models_{\mathbb{Z}} \mathcal{S}_n$ iff $G(C) \models_{\mathbb{Z}_{2^{2n}}} \mathcal{S}_n$.

C.4 D-Gröbner bases

The question whether a circuit fulfills a given specification can be answered by an ideal membership test. The theory of Gröbner bases [25] offers a decision procedure for this problem. For the polynomial rings applied in Sect C.2, we use the more general theory of D-Gröbner bases [9], where the coefficient ring is a *principal ideal domain* (PID). Let D be a PID.

Some facts about the theory of D-Gröbner bases are:

- Let $p, q, r \in D[X]$. We say q *D-reduces* to r w.r.t. p if there exists a monomial m' in q with $m' = m \text{lc}(p)$ and $r = q - mp$. If $m' = \text{lt}(q)$, we call this *top-D-reduction*.
- Let $q \in D[X]$ and $P \subseteq D[X]$. 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 r is *top-D-reduced* w.r.t. P .
- Let $q \in D[X]$ and $P \subseteq D[X]$ with UMLT. If $\text{lc}(p) = \pm 1$ for $p \in P$, then D-reduction of q w.r.t. P amounts to replacing every occurrence of $\text{lt}(p)$ in q by the tail of p .
- A basis P of an ideal $I \subseteq D[X]$ is called a *D-Gröbner basis* of I iff $\forall q \in I \exists p \in P : \text{lc}(p) \mid \text{lc}(q)$.
- Every ideal of $D[X]$ has a D-Gröbner basis, and there is an algorithm (Thm 10.14 of [9]) which, 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* and *G-polynomials*.

Definition C.25. Let $g_1, g_2 \in D[X]$. Assume

$$\begin{aligned} \text{lcm}(\text{lc}(g_1), \text{lc}(g_2)) &= b_1 \text{lc}(g_1) = b_2 \text{lc}(g_2) \text{ with } b_i \in D \text{ and} \\ \text{lcm}(\text{lt}(g_1), \text{lt}(g_2)) &= s_1 \text{lt}(g_1) = s_2 \text{lt}(g_2) \text{ with } s_i \in [X] \text{ and} \end{aligned}$$

lcm the least common multiple.

Further pick $c_1, c_2 \in D$ such that $c = \text{gcd}(\text{lc}(g_1), \text{lc}(g_2)) = c_1 \text{lc}(g_1) + c_2 \text{lc}(g_2)$, with gcd the greatest common divisor. Then define

$$\begin{aligned} \text{spol}(g_1, g_2) &:= b_1 s_1 g_1 - b_2 s_2 g_2 \\ \text{gp}(g_1, g_2) &:= c_1 s_1 g_1 + c_2 s_2 g_2 \end{aligned}$$

Lemma C.26. [Cor. 10.12 in [9]] A set $P \subseteq D[X]$ is a D-Gröbner basis of $\langle P \rangle$ iff 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 $\text{gp}(p_1, p_2)$ top-D-reduces to zero w.r.t. P .

Lemma C.27. [Thm. 11 in [76]] Let $p_1, p_2 \in D[X]$ 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 $\text{gp}(p_1, p_2)$ (top-)D-reduce to zero.

C.4.1 D-Gröbner bases applied to Multiplier Verification

We use Lemma C.27 to derive a D-Gröbner basis of the ideal $J(C)$. The following theorem shows that we neither have to compute S-polynomials nor G-polynomials.

Theorem C.28. Let R be a PID and $J(C) = \langle G(C) \cup B_0(C) \rangle$ be as in Def. C.18. Then $G(C) \cup B_0(C)$ is a D-Gröbner basis of $J(C)$ for $R = D$.

Proof. Since $G(C)$ has UMLT, $G(C) \cup B_0(C)$ has UMLT and thus by Lemma C.27, (top-)D-reduction of $\text{spol}(p, q)$ and $\text{gpol}(p, q)$ by $\{p, q\}$ gives the remainder zero for any choice $p, q \in G(C) \cup B_0(C)$ and by Lemma C.26 the claim follows. \square

For the multiplier circuits described in Sect C.2 we chose the polynomial rings $\mathbb{Z}_l[X]$ with $l \in \mathbb{N}$. For example for the truncated multiplier we set $l = 2^n$. Unless l is a prime, the ring \mathbb{Z}_l has zero divisors and is therefore not a PID. However the ideal membership test in the ring $\mathbb{Z}_l[X]$ can be reduced to an ideal membership test in the ring $\mathbb{Z}[X]$, and \mathbb{Z} is a PID.

Lemma C.29. *Let $l \in \mathbb{N}$ and let $I \subseteq \mathbb{Z}[X]$ be an ideal. There is a bijective correspondence from $q \in I + \langle l \rangle \subseteq \mathbb{Z}[X]$ to $[q] \in \{[p] \mid p \in I\} \subseteq \mathbb{Z}[X]/\langle l \rangle$, where $[q]$ is the equivalence class of q . Furthermore $\mathbb{Z}[X]/\langle l \rangle \cong \mathbb{Z}_l[X]$.*

Proof. The first claim follows from Prop. 4.3.a Chap. 10 of [4], with $\pi: q \mapsto [q]$. The second claim follows from the fundamental theorem of homomorphisms. \square

Lemma C.29 says that whenever we want to decide whether a polynomial $q \in I \subseteq \mathbb{Z}_l[X]$ we can instead check whether $q \in I + \langle l \rangle \subseteq \mathbb{Z}[X]$. And for the latter we have the concept of D-Gröbner bases available.

Lemma C.30. *Let C be an acyclic circuit, $l \in \mathbb{N}$. Then $G(C) \cup B_0(C) \cup \{l\}$ is a D-Gröbner basis for $I(C) + \langle l \rangle \subseteq \mathbb{Z}[X]$.*

Proof. It remains to show that for all $p \in G(C) \cup B_0(C)$ it holds that $\text{spol}(p, l)$ D-reduces to zero and $\text{gpol}(p, l)$ top-D-reduces to zero, which follows from Lemma C.27, because $\text{lc}(p) = -1$ and $\text{lt}(l) = 1$. \square

C.5 Variable elimination

D-Gröbner bases can be used as a kind of black-box for deciding the ideal membership of the circuit specification. However it was shown in [61, 79] that by simply reducing the specification by the polynomials $G(C) \cup B_0(C)$, the size of the intermediate reduction results increases drastically.

In [61] we presented a theorem which allows us to simplify local parts of Gröbner bases over fields without changing the rest of the circuit. We extracted specific patterns of the original multiplier, eliminated the internal variables and only the corresponding specification of the sub-circuit remained. In this work we present a more general elimination procedure which subsumes our proposed rewriting methods of [61]. We introduce a technical theorem in the fashion of Thm. 4 of [61], which is applicable in more general polynomial rings.

Definition C.31. Let $I \subseteq D[X] = D[Y, z]$ be an ideal. The ideal $I \cap D[Y]$ of $D[Y]$ is called an *elimination ideal* of I .

In general computing a D-Gröbner basis for the elimination ideal means that we explicitly need to compute a D-Gröbner basis w.r.t. a different term order. However if the D-Gröbner basis of I has UMLT, we will show that we can instantly obtain a D-Gröbner basis for the elimination ideal.

Definition C.32. Let $P \subseteq D[X]$ be a D-Gröbner basis of $\langle P \rangle$ with UMLT. We say P is *reduced for z* if the variable $z \in X$ is contained in exactly one polynomial $p \in P$ and $\text{lt}(p) = z$.

Lemma C.33. Let P be a D-Gröbner basis with UMLT and let $p \in P$. Let $H = P$ be such that all polynomials $h \neq p \in H$ are D-reduced w.r.t. p . Then H is reduced for $\text{lt}(p) = z$.

Proof. Let $f \neq p \in P$ be an arbitrary polynomial containing z . By definition $\text{lt}(f) \neq z$. Let r be the remainder of D-reducing f w.r.t. p . Because $\text{lc}(p) \in D^\times$, r is free of z .

Let $H = (P \setminus \{f\}) \cup \{r\}$. Since $\text{lm}(r) = \text{lm}(f)$ it follows that H is a D-Gröbner basis with UMLT. After repeating the steps for all $f \in P$, p is the only polynomial containing z . \square

Theorem C.34. Let $I \subseteq D[X]$ be an ideal. Let P be a D-Gröbner basis of I with UMLT which is reduced for z . Let $p \in P$ be the polynomial with $\text{lt}(p) = z$. Then $P \setminus \{p\}$ is a D-Gröbner basis with UMLT for the ideal $J = I \cap D[X \setminus \{z\}]$.

Proof. We show that $\forall f \in J \exists q \in P \setminus \{p\} : \text{lm}(q) \mid \text{lm}(f)$. Since P is a D-Gröbner basis there exists a polynomial $h \in P$ such that $\text{lm}(h) \mid \text{lm}(f)$. Because J is free of z , it follows that $\text{lm}(p) \nmid \text{lm}(f)$. Thus $h \neq p$ and consequently $h \in P \setminus \{p\}$. \square

In our approach we apply variable elimination for circuits C as follows. Let $Y = X \setminus \{z\}$. We successively select variables $z \in X$ for elimination and rewrite $G(C) \cup B_0(C)$ according to Lemma C.33 such that $G(C) \cup B_0(C)$ is reduced for z . By Thm. C.34 removing the polynomial p with $\text{lt}(p) = z$ yields a D-Gröbner basis $(G(C) \setminus \{p\}) \cup B_0(C)$ for $I(C) \cap D[Y]$.

We derive by the proof of Lemma C.30 that $(G(C) \setminus \{p\}) \cup B_0(C) \cup \{l\}$ is a D-Gröbner basis for $(I(C) + \langle l \rangle) \cap \mathbb{Z}[Y]$.

C.6 Combining SAT and Computer Algebra

In this section we present how to combine the algebraic verification approach with SAT to successfully verify complex multiplier circuits given as And-Inverter-Graphs (AIG) [70].

Multipliers can be decomposed into three components [85], cf. Fig. C.1. In the first component *partial product generation* (PPG) the partial products $a_i b_j$ (as contained in \mathcal{L}) are generated. This can for example be achieved using quadratically many AND-gates or using a more complex Booth encoding.

The second component *partial product accumulation* (PPA) reduces the partial products to two layers, where full- and half-adders are arranged in different patterns to sum

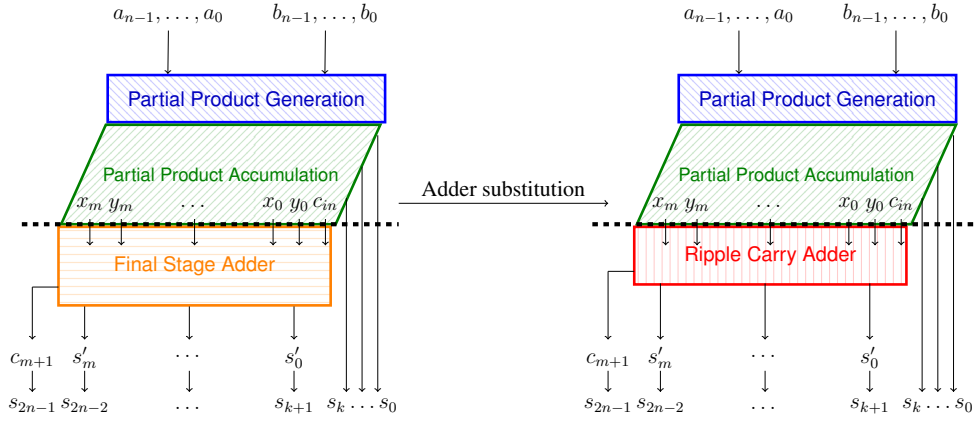


Figure C.1: Substituting the final stage adder to gain a simplified multiplier.

up the partial products. Well-known accumulation structures are array accumulation, Wallace trees or compressor trees.

In the *final stage adder* (FSA) the output of the circuit is computed. Generally adder circuits can be divided into two groups, either the carries are computed alongside the sum bits or they are calculated before the sum. Adders of the first group are usually based on a sequence of half- and full-adders, which gives them a simple but inefficient structure. Examples are ripple-carry adders 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 also called generate-and-propagate (GP) adders and examples are carry look-ahead adders and Kogge-Stone adders.

Adders of the second group are hard to verify using the algebraic approach, due to the OR trees needed to precompute carries. Due to the polynomial representation of OR-gates, cf. (C.2), this leads to an exponential blow-up in the polynomial reduction. During preparation for the SAT Race 2019 [67] we observed that checking the equivalence of different adder circuits is rather trivial for SAT solvers. We use this observation in the verification procedure and determine whether the final stage adder is a GP adder. Figure C.2 shows our tool chain used for verifying (left side) and certifying (right side) multiplier circuits given as an And-Inverter-Graph (AIG) [70]. We implemented a new dedicated reduction engine AMULET in C. Adder substitution is automatically applied and, if necessary, a simplified AIG and miter encoded as propositional formula in conjunctive normal form (CNF) are returned.

Detecting GP adders in non-synthesized multipliers is a simple task and the pseudo-code is listed in Alg. 6. In a GP adder with inputs $x_0, \dots, x_m, y_0, \dots, y_m$ and outputs $s'_0, \dots, s'_m, c_{m+1}$, cf. Fig C.1, the outputs 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 as $c_i = (x_{i-1} \wedge y_{i-1}) \vee (c_{i-1} \wedge p_{i-1})$. The precise computation of the carries c_i (recursively, unrolled or mixed) depends on the circuit architecture.

If a multiplier contains a GP adder, the most significant output bit s_{2n-1} is the carry output of the adder, i.e., $s_{2n-1} = c_{m+1}$ in Fig. C.1. Thus the loop in line 2 of Alg. 6

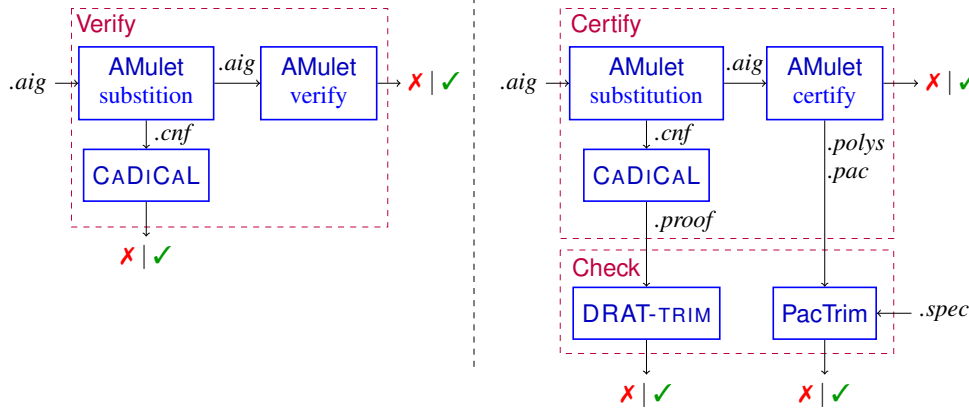


Figure C.2: Tool chain for verification (left) or certification and checking (right).

Algorithm 6: Identifying GP adders in AMULET

Input :Circuit C in AIG format
Output :Determine whether C might contain a GP adder

```

1  $j \leftarrow 2n - 2, \tau \leftarrow 1;$ 
2 while  $\tau$  and  $j \geq 0$  do
3    $\tau, c_j, p_j \leftarrow \text{Check-if-XOR-and-Identify-}p_j\text{-and-}c_j(s_j);$ 
4    $x_j, y_j \leftarrow \text{Declare-Adder-Inputs}(p_j, \tau);$ 
5    $j \leftarrow j - 1;$ 
6  $c_{in} \leftarrow c_j;$ 
7 for  $i \leftarrow j$  to  $2n - 1$  do
8    $m \leftarrow \text{Follow-and-Mark-Paths}(s_i);$ 
9 return  $m = 0$ 

```

starts at $2n - 2$. At first we check whether the output bit s_i is an XOR-gate, which can easily be identified in AIGs. If s_i is an XOR gate, its inputs are p_i and c_i . We can clearly identify which is which, because p_i has to be an XOR gate, whereas c_i cannot be an XOR gate. In the next step of the loop (line 4) we mark the inputs of the XOR gate p_i as adder inputs x_i and y_i .

If s_i is not an XOR gate or we cannot clearly identify p_i and c_i we stop the loop (indicated by τ), because then s_i is not computed by the GP adder. As can be seen in Fig. C.1, some smaller output bits are directly computed in the PPA step. We mark the smallest c_i as the carry-in c_{in} of the GP adder.

In the next phase of our algorithm we follow all input paths of s_i for $j \leq i \leq 2n - 1$. We now include s_{2n-1} , because it is the carry-out of the adder. We mark the gates alongside the paths and stop whenever we reach a marked input x_i or y_i or c_{in} . If we encounter a path, which ends at the primary inputs a_i, b_i of the multiplier, then we do not consider the final stage adder as a GP adder.

If we detect a final stage GP adder, we substitute it by a simple ripple-carry adder, which has the same inputs x_i, y_i and c_{in} . We do not change the first two stages of a

multiplier, as depicted in Fig. C.1. To prove that the ripple-carry adder is equivalent to the GP adder we generate a bit-level miter in conjunctive normal form, which is verified by a SAT solver (CaDiCaL [15]). If the final stage adder is not a GP adder we do not apply adder substitution.

Algorithm 7: Outline of verification flow in AMULET

Input : Substituted circuit C in AIG format
Output : Determine whether C is a multiplier

```

1 for  $i \leftarrow 0$  to  $2n - 1$  do
2    $S_i \leftarrow \text{Define-Cone-of-Influence}(i)$ ;
3    $\text{Order}(S_i)$ ;
4    $\text{Search-for-Booth-Encoding}(S_i)$ ;
5    $\text{Local-Elimination}(S_i)$ ;
6  $\text{Global-Elimination}()$ ;
7  $C_0 \leftarrow \text{Incremental-Reduction}()$ ;
8 return  $C_0 = 0$ 

```

After substitution we verify or certify the rewritten AIG in AMULET. The outline of the flow is depicted in Alg. 7.

For verification we use our incremental column-wise verification algorithm of [61]. The goal is to split the verification approach into smaller more manageable sub-problems by partitioning the circuit into column-wise slices and by splitting the word-level specification of a multiplier into multiple polynomials which relate the partial products, incoming carries, the sum output bit and the outgoing carries of each slice. The incremental specification presented in [61] is tailored to unsigned multipliers, but it can easily be adapted to more general multiplier specifications by adding coefficients.

We first define slices based on the input cones of the outputs and order the variables in the slices according to their level seen from the circuit inputs (line 2 in Alg. 7). This ensures that the variables are topologically sorted and the corresponding polynomials have UMLT and thus form a D-Gröbner basis.

After sorting we apply syntactic pattern matching to detect whether the circuit uses Booth encoding. In Booth encoding consecutive primary multiplier inputs are used as inputs of XOR-gates which are then combined to form an OR-gate. These XOR- and OR-gates are input to several gates in multiple slices and to increase cancellation of common monomials, we identify corresponding variables.

For local variable elimination (line 4) we loop over the gate polynomials in each slice and eliminate the variables of the leading terms which only occur in polynomials in the same slice and which are contained in exactly one other polynomial inside the slice. We repeatedly apply variable elimination until all variables of leading terms are either contained in the tails of multiple polynomials or occur in polynomials of bigger slices.

After reducing the number of variables inside the slices we eliminate variables which we marked in line 3 of Alg. 7. The difference to local variable elimination is that we now have to consider all polynomials from the circuit.

After variable elimination we apply Alg. 2 of [61] and reduce the column-wise

specification by the rewritten sliced D-Gröbner bases and report whether the final result is zero or not. Our tool AMULET uses the UMLT property of the D-Gröbner basis for D-reduction, making it much more efficient than the computer algebra systems [38, 102] used in our previous work, which are designed for more general sets of polynomials. We use the property that every leading monomial contains at most one variable with exponent 1 and with coefficient -1 and thus D-reduction reduces to replacing every occurrence of the leading variable by the tail of the polynomial. As a further optimization we employ reduction by the boolean value constraints implicitly. Whenever a term in the intermediate reduction results contains an exponent larger than 1, we immediately eliminate the exponent, without applying explicit reduction by the corresponding boolean value constraint.

If we want to certify verification we generate PAC proofs [90] in AMULET as by-product of the verification algorithm. These proofs can be checked by our independent proof checker PACTRIM [90], cf. right side of Fig. C.2. We write proofs as sequences, where each rule is of the following form:

$$\begin{array}{ll}
 + : p_i, p_j, p_i + p_j; & \begin{array}{l} p_i, p_j \text{ appearing earlier in the proof} \\ \text{or are contained in } G(C) \\ \text{and } p_i + p_j \text{ being reduced by } B(X) \end{array} \\
 * : p_i, q, qp_i; & \begin{array}{l} p_i \text{ appearing earlier in the proof} \\ \text{or is contained in } G(C) \\ \text{and } q \in R[X] \text{ being arbitrary} \\ \text{and } qp_i \text{ being reduced by } B(X) \end{array}
 \end{array}$$

These rules model the properties of an ideal, given in Def. C.6. As for verification we do not explicitly write down proof rules when reducing a boolean value constraint. In addition we extend proof rules by a deletion information, similar to clause deletion in [49]. We changed the proof checker PACTRIM accordingly. Because of Thm. C.11 and Thm. C.15 the soundness and completeness arguments given in [90] can be generalized to polynomial rings over commutative rings with unity.

Our tool PACTRIM validates the proof that the simplified AIG fulfills the given specification \mathcal{L} by checking that \mathcal{L} is derived and the derivation only uses valid proof rules. In addition we also check with DRAT-TRIM [99] the proofs generated by CADICAL for the CNF miter.

C.7 Experiments

In our experiments we used an Intel Xeon E5-2620 v4 CPU at 2.10GHz (with turbo-mode disabled) with memory limit of 128 GB. The time is listed in rounded seconds (wall-clock time) and we measure the time from starting the tools until the tools are finished or we reach the time or memory limit. The source code, benchmarks and experimental data are available at <http://fmv.jku.at/amulet>.

In our experiments we aim to provide the most comprehensive comparison by considering all different multiplier architectures used in the current state-of-the-art [80].

These benchmarks are generated with the Arithmetic Module Generator [53], which is able to generate signed and unsigned integer multipliers up to bit-width 64. We only have access to truncated multipliers using SMT models, which we generated with Boolector [83]. Additionally we generated benchmarks of large multipliers with GenMul [81] (which only scales up to 512 bits), Boolector [83] and generator scripts by Arist Kojevnikov [52]. The multiplier architectures of [52, 83] are very simple architectures without any optimizations.

In the experiments presented in Table C.1 we verify and certify different unsigned (u), signed (s) and truncated (t) multiplier architectures of 64 input bit-width. Due to shortage of space we do not present experiments of smaller bit-widths. The time out for the experiments in this table is set to 3600 sec (1h).

We show the effect of our contributions by either omitting adder substitution and using only polynomial reduction for verification (“nosub”), omitting variable elimination (“noelim”) or using the polynomial ring $\mathbb{Z}[X]$ instead of $\mathbb{Z}_l[X]$ (“nomod”). Each of the optimizations has a large effect and nearly every multiplier architecture, despite the clean multiplier architecture “sp-ar-rc”, produces a time out in one of the three columns. For truncated multipliers we would get a wrong result for “nomod”, which is marked by “NA₃”.

In the block “Verify” we measure the time for applying the tool chain as shown in the left side of Fig. C.2. We list the times AMULET needs for adder substitution (“sub”) and for verifying (“aig”) as well as the time CADICAL uses to verify the CNF miter (“cnf”). Column “tot” lists the total time.

We compare our verification results to the most recent related works [31, 80, 89]. We want to highlight that the tool of [80] is not yet available, but it enhances the approach of [79]. Thus we list the experiments of their work [80], which are run on an Intel Xeon E3-1270 v3 CPU with 3.50 GHz and thus is a slightly faster CPU than ours. Experiments which are not available for comparison are marked by “NA₂”.

The tool of [31] uses a certain optimization “&atree”. After contacting the first and last authors of [106] we were told that this option only works for simple multipliers. Using this flag on more complex multipliers leads to incompleteness, which we mark again by “NA₃”. If “&atree” is omitted, all experiments produce a segmentation fault.

It can be seen that in contrast to our previous work [89], we are able to verify all benchmarks within seconds and we are an order of magnitude faster than the currently most successful approach of [80]. The tools of related work are only partially applicable to verify signed and truncated multipliers, because the specification used in these tools is fixed to (un)signed multiplier circuits. We mark non-applicability with “NA₁”.

In “Certify” and “Check” we present the time used for certifying and checking, cf. right side of Fig. C.2. The columns of “Certify” have the same form as “Verify”. In “Check” we list the times DRAT-TRIM (“cnf”) [99] and PACTRIM (“aig”) [90] need for proof checking. The column “total” lists the total time used to certify and check the multipliers. Certifying a multiplier is around twice as slow than verifying, because additional polynomial operations are necessary to match the proof rules. In the last two columns we present the sizes of the proofs. The proof size of CNFs is measured by the number of added RUP clauses [47]. A size of 0 means, that the final stage adder is not a

Table C.1: Verification and Certification Time. \mathcal{L} column: unsigned (u), signed (s) or truncated (t) multiplier specification.

architecture	n	\mathcal{L}	nosub	nomod	noelim	Verify				[80]	[31]	[89]	Certify				Check			total	proof size	
						sub	cnf	aig	tot				sub	cnf	aig	tot	cnf	aig	tot		cnf	aig
sp-ar-rc	64	u	1	1	2	0	0	1	1	NA ₂	0	133	0	0	2	2	0	3	3	5	0	188 290
sp-dt-lf	64	u	TO	1	3	0	0	2	2	31	NA ₃	TO	0	0	2	3	0	3	3	6	34 423	186 170
sp-wt-cl	64	u	TO	TO	3	0	9	1	11	96	NA ₃	TO	0	9	2	12	7	3	10	21	264 471	191 623
sp-bd-ks	64	u	TO	TO	2	0	1	1	3	162	NA ₃	TO	0	2	2	4	1	3	4	8	78 567	190 915
sp-ar-ck	64	u	TO	1	2	0	0	1	1	143	NA ₃	TO	0	0	2	2	0	3	3	5	1 432	187 251
bp-ar-rc	64	u	1	TO	118	0	0	1	1	53	NA ₃	TO	0	0	2	2	0	3	3	5	0	161 815
bp-ct-bk	64	u	TO	TO	100	0	0	1	2	119	NA ₃	TO	0	0	2	2	0	3	3	5	27 552	138 179
bp-os-cu	64	u	2	TO	TO	0	0	2	2	95	NA ₃	TO	0	0	3	3	0	4	4	7	0	166 967
bp-wt-cs	64	u	1	TO	114	0	0	1	1	75	NA ₃	TO	0	0	2	2	0	3	3	6	0	161 747
sp-ar-rc	64	s	1	1	2	0	0	1	1	NA ₁	0	NA ₁	0	0	2	2	0	3	3	6	0	188 426
bp-wt-cl	64	s	TO	3	109	0	10	1	11	NA ₁	NA ₃	NA ₁	0	10	2	12	7	3	10	22	261 650	151 355
btor	64	t	0	NA ₃	1	0	0	0	1	NA ₁	NA ₁	NA ₁	0	0	1	1	0	1	1	2	0	70 374

NA₁: tool not applicable to type \mathcal{L} NA₂: tool not yet availableNA₃: incompleteness (see text)

TO: 3600 sec

Benchmarks are either generated by the Arithmetic Module Generator of [53] or by Boolector [83] (btor).

PPG: simple (sp), Booth (bp) PPA: Dadda tree (dt), Wallace tree (wt), balanced delay tree (bd), array (ar), compressor tree (ct), overturned-stairs tree (os)

FSA: Ladner-Fischer (lf), carry look-ahead (cl), Kogge-Stone (ks), carry-skip (ck), ripple-carry (rc), Brent-Kung (bk), conditional sum (cu), carry select (cs)

Table C.2: Verifying benchmarks of large input size. \mathcal{L} column: unsigned (u) multiplier specification.

architecture	n	\mathcal{L}	Verify				[80]	[31]	AIG size
			sub	cnf	aig	tot			
btor	128	u	0	0	9	10	NA ₂	2	123 k
kjvkv	128	u	0	0	9	9	NA ₂	2	195 k
sp-ar-rc	128	u	0	0	10	10	349	2	195 k
sp-dt-lf	128	u	0	2	13	15	490	NA ₃	195 k
sp-wt-bk	128	u	0	1	18	20	746	NA ₃	198 k
btor	256	u	1	0	119	120	NA ₂	19	522 k
kjvkv	256	u	1	0	84	86	NA ₂	18	782 k
sp-ar-rc	256	u	1	0	84	86	8 720	20	782 k
sp-dt-lf	256	u	3	6	164	174	12 874	NA ₃	780 k
sp-wt-bk	256	u	3	3	170	177	21 454	NA ₃	790 k
btor	512	u	7	0	968	975	NA ₂	300	2 093 k
kjvkv	512	u	9	0	774	783	NA ₂	247	3 138 k
sp-ar-rc	512	u	10	0	770	780	192 640	312	3 138 k
sp-dt-lf	512	u	25	21	1 539	1 585	240 051	NA ₃	3 133 k
sp-wt-bk	512	u	24	9	1 560	1 594	492 320	NA ₃	3 157 k
btor	1024	u	97	0	10 623	10 720	NA ₂	8 323	8 379 k
kjvkv	1024	u	106	0	5 463	5 570	NA ₂	3 778	12 567 k
btor	2048	u	1 026	0	89 565	90 591	NA ₂	150 976	33 536 k
kojvkv	2048	u	1 057	0	67 733	68 790	NA ₂	74 514	50 299 k

NA₂: tool not yet availableNA₃: incompleteness (see text)

Benchmarks generated by Boolector [83] (btor), from [52] (kjvkv) and [81].

PPG: simple (sp) PPA: array (ar), Dadda tree (dt), Wallace tree (wt)

FSA: ripple-carry (rc), Ladner-Fischer (lf), Brent-Kung (bk)

GP adder. Thus a trivial CNF is reported which does not yield a resolution proof. The size of the algebraic proofs is measured by the number of PAC rules [90].

In the experiments of Table C.2 we list the time to verify large multiplier designs. We are able to verify multipliers of input size 2048, consisting of more than 50 million AIG nodes in around 19h. Certifying and checking these benchmarks is around three times slower. For example certifying “kojvkv-2048” needs 34h wall-clock time. Checking the (uncompressed) proof, which has a size of 1.4 TB, needs 20h.

C.8 Conclusion

In this paper we combine SAT and computer algebra to verify large unsigned, signed and truncated integer multipliers. Our theory describes polynomial reasoning over more general rings. We formulate and prove soundness and completeness.

We show how modular reasoning can be simulated by integer reasoning and revisit and apply existing D-Gröbner bases theory from the literature. Modular arithmetic is required to specify truncated multipliers. It also improves performance substantially. We formalize variable elimination too.

Our main contribution consists of extracting complex final stage adders, which are substituted by simple adders. Correctness of this substitution is proven by SAT and correctness of the simplified multiplier by our dedicated reduction engine.

Our experiments show that the combination of these ideas allow us to scale up verification to large multipliers of 2048 bits. We are also able to verify complex multipliers an order of magnitude faster than the previous state-of-the-art. Furthermore, we produce proof certificates in contrast to other approaches. These proofs are checked independently to validate the verification results.

In future work we want to apply our approach to synthesized multipliers where technology mapping is applied and to other arithmetic circuits beyond integer multipliers. Another intriguing research direction is to integrate both polynomial and clausal reasoning in a common proof format.



Paper D

SAT, Computer Algebra, Multipliers

Published as invited paper in the Post-Proceedings of the 5th and 6th Vampire Workshops, Vampire 2018 and Vampire 2019, pages 1–18, Lisbon, Portugal, 2019.

Authors Daniela Kaufmann, Armin Biere and Manuel Kauers.

Acknowledgement This work is supported by the Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), P31571-N32, SFB F5004, LIT AI Lab funded by the state of Upper Austria.

Abstract Verifying multiplier circuits is an important problem which in practice still requires substantial manual effort. The currently most effective approach uses polynomial reasoning. However parts of a multiplier, i.e., complex final stage adders are hard to verify using computer algebra. In our approach we combine SAT and computer algebra to substantially improve automated verification of integer multipliers. In this paper we focus on the implementation details of our new dedicated reduction engine, which not only allows fully automated adder substitution, but also employs polynomial reduction efficiently. Our tool is furthermore able to generate proof certificates in the practical algebraic calculus and we also investigate the size of these proofs for one specific multiplier architecture.

D.1 Introduction

Formal verification of arithmetic circuits is extremely important to help to prevent issues like the famous Pentium FDIV bug. There have been many attempts since then to verify such circuits, but even today the problem of formally verifying arithmetic circuits, and especially multiplier circuits, is still considered to be hard and cannot be applied fully automated. In principle, theorem provers in combination with SAT are able to certify industrial multipliers [54]. However, such approaches lack automation.

Currently the most successful automated approach uses polynomial reasoning [31,61,79,80,106] and in recent years has seen significant progress. The approach of [79,80] employs local cancellation of vanishing monomials in converging cones, which allows verifying a large variety of multiplier architectures much more efficiently than previous work. The authors of [31,106] eliminate redundant polynomials by identifying full- and half-adders in the multipliers. This technique is able to verify large simple multipliers, but fails on even slightly more complex multiplier architectures.

In our method [62] we combine two approaches, i.e., SAT and computer algebra. We observe that final stage adders of multipliers are a real challenge for the algebraic approach as some adder designs rely on sequences of OR-gates, which lead to an explosion of the polynomial representation of the intermediate results. Contrarily SAT solvers can easily verify the equivalence of adder circuits. Therefore we apply adder substitution and replace complex final stage adders by simpler adders and verify the correctness of the substitution using SAT solvers. The correctness of the simplified multiplier is shown using computer algebra. Our method is an order of magnitude faster than related work and is able to verify circuits with input bit-width 2048.

Our reduction engine AMULET [62] detects complex final stage adders and applies adder substitution fully automatically. A bit-level miter in conjunctive normal form (CNF) as well as a rewritten multiplier is generated. In the verification phase AMULET uses the structure of the polynomial representation of circuits and thus is more efficient in circuit verification than computer algebra systems [38,102] used in our previous work. Additionally we apply preprocessing based on variable elimination.

Furthermore AMULET efficiently produces certificates in the PAC format [90], which allow checking the correctness of the verification results. None of the related work [31,79,80,106] produces certificates.

This paper provides supplementary material for an invited talk at Vampire'19 of the second author based on [61,62]. We discuss the implementation of AMULET and present the underlying algorithms of [62] in more detail. Additionally we provide a generalization of our incremental approach of [61]. We further show that we are able to generate proof certificates of quadratic length for simple multipliers.

D.2 Algebraic approach

We consider acyclic gate-level circuits C which implement integer multiplication. The circuits have $2n$ input bits $a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1} \in \{0, 1\}$, as well as $2n$ output bits $s_0, \dots, s_{2n-1} \in \{0, 1\}$ and further a number of internal logical gates denoted by $g_0, \dots, g_k \in \{0, 1\}$. Let R be a commutative ring with unity and let $R[X]$ be the polynomial ring over R and the set of variables

$$X = \{ a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1}, s_0, \dots, s_{2n-1}, g_0, \dots, g_k \}.$$

A *term* $\tau = x_1^{d_1} \dots x_r^{d_r}$ is a product of powers of variables for certain $d_1, \dots, d_r \in \mathbb{N}$. A *monomial* is a multiple of a term $c\tau$, with $c \in R$ and a *polynomial* p is a finite sum of monomials.

An order \leq is fixed on the set of terms such that $1 \leq \tau$ and $\sigma_1 \leq \sigma_2 \Rightarrow \tau\sigma_1 \leq \tau\sigma_2$ for all terms τ, σ_1, σ_2 . Such an order is a *lexicographic term order* if for all terms $\sigma_1 = x_1^{d_1} \cdots x_r^{d_r}$, $\sigma_2 = x_1^{e_1} \cdots x_r^{e_r}$ it holds that $\sigma_1 < \sigma_2$ iff there exists i with $d_j = e_j$ for all $j < i$, and $d_i < e_i$. The largest term (w.r.t. \leq) in a polynomial $p = c\tau + \cdots$ is called the *leading term* $\text{lt}(p) = \tau$. The *leading coefficient* and *leading monomial* of p are defined accordingly. Furthermore we call $p - c\tau$ the *tail* of p .

The *specification* of a circuit describes the desired relation between the outputs and inputs of a circuit. If for all possible inputs the circuit computes the desired output, we say that the circuit fulfills its specification and thus is correct. Formal verification aims to derive whether a circuit fulfills its specification or not. In the algebraic verification approach we model each logical gate by a polynomial. Correctness of the circuit is shown by deriving that the specification, also encoded as a polynomial \mathcal{L} , is implied by the gate polynomials.

The polynomial ring R is fixed with the specification. Although we model integer multiplication, we showed in [62] that it is beneficial to use more general polynomial rings which allow modular reasoning.

Definition D.1. The specification \mathcal{U}_n of n -bit unsigned integer multipliers in the ring $\mathbb{Z}_{2^{2n}}[X]$ is given as

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

As discussed in [62] modular reasoning also allows us to define the specification of truncated multipliers, i.e., a truncated multiplier only returns the n least significant output bits.

Definition D.2. The specification \mathcal{T}_n of n -bit truncated multipliers in the ring $\mathbb{Z}_{2^n}[X]$ is given as

$$\mathcal{T}_n = \sum_{i=0}^{2n-1} 2^i s_i - \left(\sum_{i=0}^{n-1} 2^i a_i \right) \left(\sum_{i=0}^{n-1} 2^i b_i \right) = \sum_{i=0}^{n-1} 2^i s_i - \sum_{i=0}^{n-1} \sum_{j=0}^{n-1-i} 2^{i+j} a_i b_j. \quad (\text{D.2})$$

After fixing the specification and thus the coefficient ring R , each logical gate in the circuit is encoded as a polynomial, such as:

$$\begin{aligned} u = \neg v & \quad \text{implies} \quad 0 = -u + 1 - v \\ u = v \vee w & \quad \text{implies} \quad 0 = -u + v + w - vw \\ u = v \wedge w & \quad \text{implies} \quad 0 = -u + vw \\ u = v \oplus w & \quad \text{implies} \quad 0 = -u + v + w - 2vw. \end{aligned} \quad (\text{D.3})$$

The polynomial equations in (D.3) are chosen in such a way that the possible solutions with $u, v, w \in \{0, 1\}$ of the polynomials in $R[X]$ are the solutions of the gate constraints and vice versa. Note, the polynomials above are defined in the ring $\mathbb{Z}[X]$ and thus the structure may differ for different coefficient rings R . We order the terms according to a *reverse topological lexicographic term order*, such that the output variable of a gate is always greater than the variables attached to the input edges of that gate.

Definition D.3. By $G(C) \subseteq R[X]$ we denote the set of *circuit polynomials* which contains for each gate of C the corresponding polynomial of (D.3). We further have *Boolean value constraints* $x(1 - x) = 0$ for $x \in X$, encoding that x is a Boolean variable. Let $B(Y) = \{y(1 - y) \mid y \in Y\} \subseteq R[X]$ for $Y \subseteq X$, be the set of Boolean value constraints for Y .

Definition D.4. A nonempty subset $I \subseteq R[X]$ is called an *ideal* if $\forall p, q \in I : p + q \in I$ and $\forall p \in R[X] \forall q \in I : pq \in I$. A set $P = \{p_1, \dots, p_s\} \subseteq R[X]$ is called a *basis* of I if $I = \{p_1q_1 + \dots + p_sq_s \mid q_1, \dots, q_s \in R[X]\}$. We say I is generated by P and write $I = \langle P \rangle$. The sum of two ideals I and J is defined as $I + J = \{p + q \mid p \in I, q \in J\}$.

In [62] we showed that the question whether \mathcal{L} is implied by the gate polynomials of C and the Boolean value constraints can be answered by deciding a so-called ideal membership problem: “Given $q \in R[X]$ and a (finite) set of polynomials $P \subseteq R[X]$, decide whether $q \in \langle P \rangle$.”

Definition D.5. Let $P \subseteq R[X]$. If for a certain term order, all leading terms of P only consist of a single variable with exponent 1 and are unique and further all leading coefficients are multiplicatively invertible in R , 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 $B_0(P) = B(X_0(P))$.

It is easy to see that for an acyclic circuit C the set $G(C)$ has UMLT for a fixed reverse topological term order. Further $X_0(P)$ contains only circuit inputs a_i, b_i .

Definition D.6. Let C be a circuit and let $J(C) = \langle G(C) \cup B_0(C) \rangle \subseteq R[X]$, with $B_0(C) = B_0(G(C))$.

Corollary D.7. [62] A circuit C fulfills \mathcal{L} iff $\mathcal{L} \in J(C)$.

The theory of Gröbner bases [25] offers a decision procedure for the ideal membership problem. For our purpose we use the more general theory of D-Gröbner bases [9], where the coefficient domain D is a principal ideal domain (PID). Let $p, q, r \in D[X]$ and let $P \subseteq D[X]$. A basis P of an ideal $I \subseteq D[X]$ is a *D-Gröbner basis* of I iff $\forall q \in I \exists p \in P : \text{lm}(p) \mid \text{lm}(q)$. Every ideal of $D[X]$ has a D-Gröbner basis, and there is an algorithm (Thm. 10.14 of [9]) which, given an arbitrary basis of an ideal, computes a D-Gröbner basis of it in finitely many steps.

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 $r = q - mp$. 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 P is a D-Gröbner basis, then $r = 0$ iff $q \in \langle P \rangle$.

For the specifications listed in Def. D.1 and Def. D.2 we fixed the polynomial rings to $\mathbb{Z}_l[X]$ for $l \in \mathbb{N}$. In general \mathbb{Z}_l is not a PID, but we showed in [62] that the ideal membership problem in $\mathbb{Z}_l[X]$ can be converted to an ideal membership problem in the ring $\mathbb{Z}[X]$, with \mathbb{Z} being a PID. Whenever we want to decide whether a polynomial $q \in I \subseteq \mathbb{Z}_l[X]$ we can instead check whether $q \in I + \langle l \rangle \subseteq \mathbb{Z}[X]$. For the latter we have the concept of D-Gröbner bases available. And since $G(C)$ has UMLT we can directly derive a D-Gröbner basis for $J(C) + \langle l \rangle$.

Lemma D.8. [62] Let $l \in \mathbb{N}$. Then $G(C) \cup B_0(C) \cup \{l\}$ is a D -Gröbner basis for $J(C) + \langle l \rangle \subseteq \mathbb{Z}[X]$.

D.2.1 Incremental Verification

In [61] we introduced an incremental verification algorithm, which splits the verification problem into smaller more manageable subproblems by partitioning the circuit into column-wise slices and splitting the word-level specification into multiple smaller specifications which relate the partial products, incoming carries, sum output bit and the outgoing carries of each slice. However this algorithm is tailored to multiplication of unsigned bit-vectors. In this section we show how to apply this procedure to different multiplier specifications. As the number of output bits varies for different multipliers, e.g., in Def. D.2, we denote the number of output bits by the constant m and fix $l = 2^m$ in this section.

Definition D.9. Let $I_i := \{\text{gate } g \mid g \text{ is in input cone of } s_i\}$ be the input cone of each output bit s_i for $0 \leq i < m$. A slice S_i is defined as the difference of consecutive cones I_i , i.e., $S_0 := I_0$ and $S_{i+1} := I_{i+1} \setminus \bigcup_{j=0}^i S_j$.

Definition D.10 (Sliced Gröbner Bases). Let $G_i(C)$ be the set of circuit polynomials of the gates in a slice S_i . The terms are ordered such that the requirements of Lemma. D.8 are fulfilled. We define by $X_0(G_i)$ the set of variables that do not occur as leading terms in $G_i(C)$ and further define $B_0(G_i) = B(X_0(G_i))$.

Corollary D.11. $G_i(C) \cup B_0(G_i) \cup \{2^m\}$ is a D -Gröbner basis for $\langle G_i(C) \cup B_0(G_i) \rangle + \langle 2^m \rangle$.

Corollary D.11 follows directly from Lemma. D.8. It is easy to see that $\langle G_i(C) \cup B_0(G_i) \rangle$ contains all the Boolean value constraints $B(G_i)$ for the gate variables in S_i , thus we may use them in the reduction process to eliminate exponents greater than 1 in the intermediate reduction results. After splitting the circuit, we are now going to split the word-level specification of a multiplier.

Definition D.12. Let C be a multiplier circuit which is sliced according to Def. D.9 and let \mathcal{L} be the specification of C . For slice S_i with $0 \leq i < m$ let $P_i = \sum_{j+k=i} \alpha_{jk} a_j b_k$ be the *partial product sum* of column i , where the constant α_{jk} is the coefficient of the term $a_j b_k$ in \mathcal{L} .

Definition D.13. Let C be a multiplier circuit. A sequence of $m + 1$ polynomials C_0, \dots, C_m over the variables of C is called a *carry sequence* if for all $0 \leq i \leq m$ it holds that

$$-C_i + C_{i+1} + \alpha_i s_i + P_i \in J(C)$$

where the constant α_i is the coefficient of s_i in \mathcal{L} . We call the polynomials $-C_i + C_{i+1} + \alpha_i s_i + P_i$ the *carry recurrence relations* for the sequence C_0, \dots, C_m .

It remains to fix the boundary polynomial C_m , where we simply choose $C_m = 0$. Our incremental algorithm is shown in Alg. 8 and it follows from the proof of Thm.6 in [61] that Alg. 8 is correct.

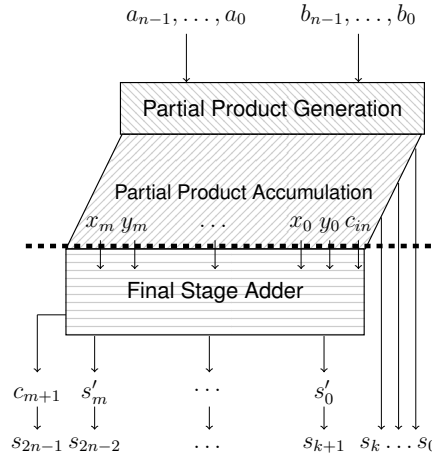


Figure D.1: The components of a multiplier.

Algorithm 8: Multiplier Checking Algorithm

Input : Circuit C with m output bits, sliced Gröbner bases G_i

Output : Determine whether C is a correct multiplier

- 1 $l \leftarrow 2^m, C_m \leftarrow 0;$
 - 2 **for** $i \leftarrow m - 1$ **to** 0 **do**
 - 3 $C_i \leftarrow \text{Remainder}(C_{i+1} + \alpha_i 2^i s_i - 2^i P_i, G_i(C) \cup B(G_i) \cup \{l\})$
 - 4 **return** $C_0 = 0$
-

D.3 SAT

Computer algebra is able to verify simple multipliers very efficiently. However more complex multiplier architectures still impose quite a challenge and lead to a monomial blow-up in the intermediate reduction results. The reason for this blow-up are certain adder structures, which are part of the multipliers. During preparation for the SAT Race 2019 [67] we observed that checking the equivalence of different adder circuits is rather trivial for SAT solvers. We make use of this observation in the verification procedure and combine computer algebra and SAT. We summarize the main idea, as presented in [62].

Generally multipliers can be decomposed into three components [85], which are shown in Fig. D.1. In the first component *partial product generation* (PPG) the partial products $a_i b_j$ as contained in \mathcal{L} are derived. This can for example be achieved using simple AND-gates or using a more complex Booth encoding. In the second stage *partial product accumulation* (PPA) the partial products are reduced to two layers using full- and half-adders. In the last stage the output of the circuit is computed using an adder circuit. Hence we call this component *final stage adder* (FSA).

Adder circuits can be split into two groups. Either the carries are computed simultaneously with the sum bits or they are calculated separately before the sum to decrease

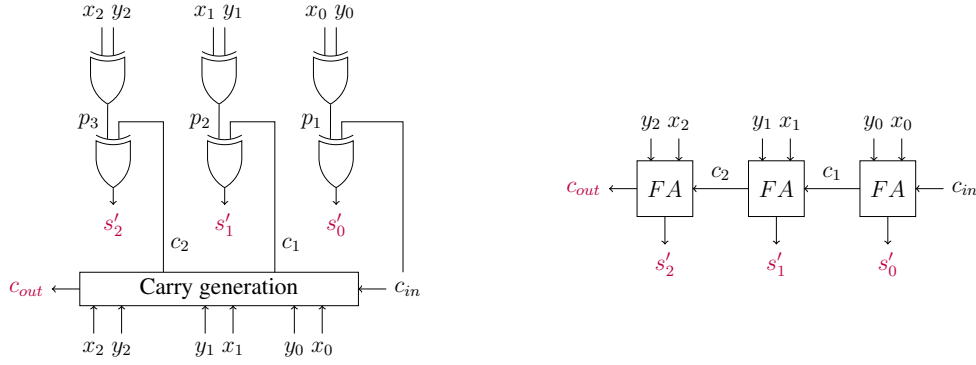


Figure D.2: GP adder (left) and equivalent RCA (right).

the latency of carry computation. A scheme for both adder types can be seen in Fig. D.2. Adders of the first group are usually based on a sequence of half- and full-adders, which gives them a simple but inefficient structure, e.g., ripple-carry adders. Adders of the second group are also called generate-and-propagate (GP) adders. In a GP adder with inputs $x_0, \dots, x_m, y_0, \dots, y_m, c_{in}$ and outputs $s'_0, \dots, s'_m, c_{out}$ 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 D.14. Let $o = o_2 \vee x_0, o_2 = o_1 \vee x_1, o_1 = x_3 \vee x_2$ represent a sequence of three OR-gates, which can be simplified to $o = x_0 \vee x_1 \vee x_2 \vee x_3$. The corresponding polynomial representation $o = x_0 + x_1 - x_0x_1 + x_2 - x_0x_2 - x_1x_2 + x_0x_1x_2 + x_3 - x_0x_3 - x_1x_3 + x_0x_1x_3 - x_2x_3 + x_0x_2x_3 + x_1x_2x_3 - x_0x_1x_2x_3$ contains $2^4 - 1$ monomials.

In our approach we identify whether the FSA is a GP adder, using the equations $s'_i = p_i \oplus c_i$ and $p_i = x_i \oplus y_i$. The algorithm is described in detail in Sect. D.4, where we present our tool AMULET. If we detect that the FSA is a GP adder, we substitute the FSA by a simple ripple-carry adder (RCA), which has the same inputs $x_0, \dots, x_m, y_0, \dots, y_m, c_{in}$ than the original FSA. We do not change the first two stages PPG and PPA. To prove that the RCA is equivalent to the GP adder we generate a bit-level miter in CNF, which 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. Figure D.3 shows the original multiplier (purple) as well as the RCA and the bit-level miter (green). The dashed boxes depict which components of the extended multiplier are verified using SAT (red) and computer algebra (blue).

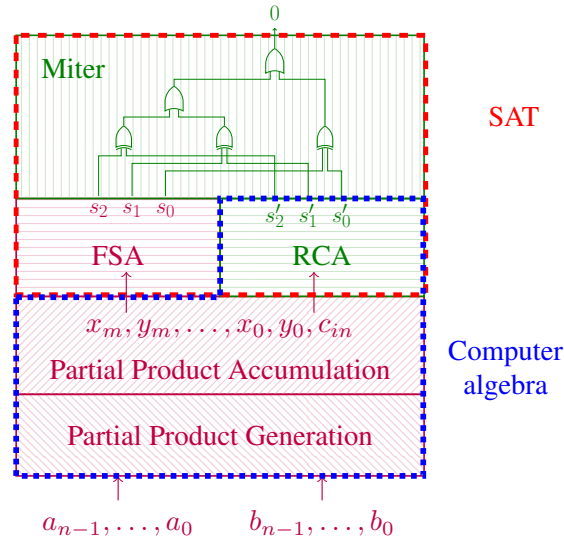


Figure D.3: Reasoning techniques used to verify the extended multiplier circuit.

D.4 AMulet

In this section we explain implementation details of our tool AMULET. Our tool, which is written in C reads multipliers given as And-Inverter-Graphs (AIG) [70] and automatically applies adder substitution and verification. Additionally we are able to generate proof certificates.

D.4.1 Adder Substitution

Our algorithm, which identifies whether the FSA is a GP adder and, if necessary, replaces the FSA by a RCA is shown in Alg. 9. It reads the original multiplier and returns a circuit in the AIG format as well as a CNF. To identify GP adders we highly relate on their structure as presented in Sect. D.3. In particular we rely on the fact that the outputs s'_i are always outputs of XOR-gates and that the carries c_i are never outputs of XOR-gates.

In the initialization phase AMULET reads the given multiplier and for each node in the AIG we introduce a unique variable. Variables in AMULET are organized in an ordered array, where the indices match the literal (divided by 2) of the AIG node. As there is a one-to-one correspondence between variables and AIG nodes we will use both terms interchangeably. We further identify whether the variable is an output or an internal gate of an XOR gate, using syntactic pattern matching.

The variable τ of line 2 acts as an error-flag. In line 3 we identify whether the output s_{2n-1} of the multiplier is the carry output c_{out} of the FSA, which is not always the case. In some multiplier architectures the output s_{2n-1} is computed as an XOR, whose inputs are the carry output c_{out} of the FSA and some output from the PPA step, which is usually again an XOR gate. Thus in line 3 we identify whether s_{2n-1} is an XOR gate. If not,

Algorithm 9: Adder substitution in AMULET

Input : Circuit C in AIG format
Output : Rewritten Circuit C' in AIG format, bit-level miter as CNF F

```

1 Init( $C$ );
2  $\tau \leftarrow 1$ ;
3  $c_{out}, \tau \leftarrow \text{Identify-Carry-Out}(s_{2n-1})$ ;
4 if  $\tau = 0$  then
5   | return  $C, 0$ ;
6  $j \leftarrow 2n - 2, \sigma \leftarrow 1$ ;
7 while  $\sigma$  and  $j \geq 0$  do
8   |  $\sigma \leftarrow \text{Check-if-XOR}(s_j)$ ;
9   |  $\sigma, c_j, p_j \leftarrow \text{Identify-p}_j\text{-and-c}_j(s_j, \sigma)$ ;
10  |  $\sigma, x_j, y_j \leftarrow \text{Mark-Adder-Inputs}(p_j, \sigma)$ ;
11  |  $j \leftarrow j - 1$ ;
12  $c_{in} \leftarrow c_j$ ;
13  $\tau \leftarrow \text{Follow-and-Mark-Paths}(c_{out}, X, Y, c_{in})$ ;
14 for  $i \leftarrow j + 1$  to  $2n - 2$  do
15   |  $\tau \leftarrow \text{Follow-and-Mark-Paths}(s_i, X, Y, c_{in}, \tau)$ ;
16 if  $\tau = 0$  then
17   | return  $C, 0$ ;
18  $R \leftarrow \text{Generate-AIG-RCA}(X, Y, c_{in})$ ;
19  $M \leftarrow \text{Generate-Miter}(C, R)$ ;
20  $F \leftarrow \text{Miter-to-CNF}(M)$ ;
21  $C' \leftarrow \text{Generate-Rewritten-AIG}(C, R)$ ;
22 return  $C', F$ 

```

then $s_{2n-1} = c_{out}$. If on the other hand s_{2n-1} is an XOR gate we examine the inputs of s_{2n-1} and identify which child is not an XOR gate. This child is then identified as c_{out} . If neither input is an XOR gate, we cannot clearly identify c_{out} of the FSA and set $\tau = 0$. In that case the algorithm terminates and returns the original multiplier and an empty bit-level miter.

In the while-loop we identify the inputs $x_0, \dots, x_m, y_0, \dots, y_m, c_{in}$ of the FSA. We do not know the concrete value of m in advance, as it depends on the multiplier architecture. Hence we recursively iterate over the outputs of the multiplier. We start the loop at the output s_{2n-2} , since s_{2n-1} was used to identify the carry output of the FSA. In line 8 we check if the output s_j is an XOR gate. If so, we identify the propagate bit p_j and the carry bit c_j in the next step. Here we rely on the fact that p_j is an XOR gate and c_j is not an XOR gate. Using p_j we mark the inputs x_j, y_j of the adder in the next step.

As shown in Fig. D.1 not all output bits of the multiplier are computed by the FSA. Smaller output bits may already be computed in the PPA step. Hence at some point we are not able to identify p_j, c_j or x_j, y_j anymore, which we capture in σ . If the FSA is not a GP adder the loop will directly stop after the first iteration. The carry-in c_{in} of the FSA is set to the smallest c_j , which was identified.

In lines 13 to 15 we mark all gates which belong to the FSA. We start at the carry

output c_{out} resp. sum outputs s_{j+1}, \dots, s_{2n-1} 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 the input variables a_i, b_i 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.

We generate an equivalent RCA in line 18. A RCA is simply a sequence of full-adders, cf. Fig. D.2 and the AIG encoding of a full-adder can be seen in Fig. D.5b. After the RCA is generated, the bit-level miter is defined. It contains all the gates which are identified to belong to the GP adder and the gates of the RCA. Furthermore we add XOR gates, whose inputs are corresponding pairs of output bits of the two adders. These XOR gates are summed up by a sequence of OR-gates, cf. Fig. D.3. If the two adders are equivalent, all equivalent pairs of output bits compute the same result. Thus the outputs of the XOR gates are 0, consequently all OR-gates are 0 and the output of the miter is 0.

The equivalence of the adders is verified using a SAT-solver. Hence the bit-level miter is translated into a CNF F in line 20. More precisely, the propositional formulas represented by each AIG node are translated into CNF, which is rather straightforward. If for example an AIG node represents $x = a \wedge \bar{b}$, the equivalent propositional formula is $\neg(x \leftrightarrow a \wedge \bar{b}) = \top$ which can be translated to the CNF $(x \vee \bar{a} \vee b) \wedge (\bar{x} \vee a) \wedge (\bar{x} \vee \bar{b}) = \top$. We iterate over each node and output the corresponding clauses in DIMACS format. The final clause which is added, is the assumption that the output of the miter is 1. Thus for a correct adder substitution the SAT solver has to return that the CNF is UNSAT.

In the rewritten multiplier, we keep all nodes of the original multiplier, which are not marked to be an element of the FSA and replace the subgraph defining the GP adder by the AIG of the RCA. The rewritten AIG C' as well as the CNF F are returned by AMULET.

D.4.2 Verification

After applying adder substitution the multiplier is verified. The pseudo-code can be seen in Alg. 10. During initialization, which is similar to Alg. 9, we fix the specification polynomial $\mathcal{L} \in \mathbb{Z}_l[X]$ and thus the constant l . As there are now more data structures involved, in particular representation of polynomials, let us briefly discuss our design decisions. The variables are organized as an ordered array. Terms are represented as ordered linked lists of variables. In general terms will be used multiple times during the reduction process, thus they are organized in a hash table. Monomials contain a coefficient and a term. Since the values of the coefficients exceed 2^{64} , we use the GMP library [44] for number representation. Polynomials are represented as sorted linked lists of monomials. In the data structure *slice* we store the gates which are assigned to a slice and the corresponding gate polynomials.

We use our incremental verification approach, cf. Alg. 8. Hence we define the slices

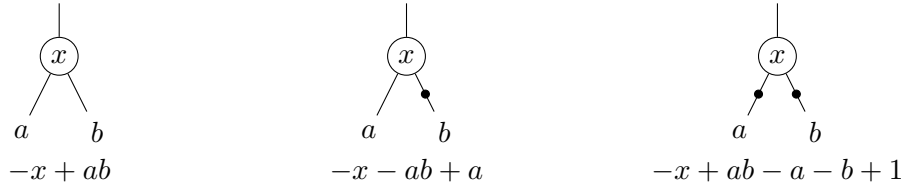
Algorithm 10: Outline of verification flow in AMULET

Input : Substituted Circuit C in AIG format
Output : Determine whether C is a multiplier

```

1  $\mathcal{L}, l \leftarrow \text{Init}(C)$ ;
2 for  $i \leftarrow 0$  to  $2n - 1$  do
3    $S_i \leftarrow \text{Define-Slices}(i)$ ;
4    $\text{Order-Slices}(S_i)$ ;
5    $G_i(C) \leftarrow \text{Init-Polynomials-of-Slices}(S_i)$ ;
6  $\Omega \leftarrow \text{Search-for-Booth-Encoding}(C)$ ;
7 for  $i \leftarrow 0$  to  $2n - 1$  do
8    $\text{Local-Elimination}(G_i(C), l)$ ;
9  $\text{Global-Elimination}(\Omega)$ ;
10  $C_0 \leftarrow \text{Incremental-Reduction}(\mathcal{L}, G_i(C))$ ;
11 return  $C_0 = 0$ 

```

**Figure D.4:** All polynomial encodings covered by AIG nodes.

as differences of consecutive input cones as introduced in Def. D.9. However in certain cases not all gates are assigned to the correct slices. To this end we *merge* and *promote* gates as described in [61]. Additionally we identify which nodes are *carry* nodes, i.e., which nodes are used as inputs of nodes in bigger slices.

After the slices are defined we fix the reverse topological lexicographic term ordering in line 4. The gates inside the slices are ordered according to their reverse topological appearance and slices are ordered in descending order. Thus also the total order of the variables is reverse topological. As a consequence the polynomials $G_i(C)$, which are introduced in line 5, automatically form a D-Gröbner basis. Each AIG node represents an AND-gate between two inputs, which may or may not be inverted. Consequently three different polynomials are possible, as can be seen in Fig. D.4. For each node we introduce the corresponding polynomial with x, a, b replaced by the corresponding variables. We further add for each output s_i a *linking polynomial* $-s_i + g_k$ to clearly mark which AIG node represents an output bit. All these polynomials mark our initial constraint set, i.e., the set $G(C)$ of Def. D.3.

We apply syntactic pattern matching to identify whether the partial products are generated using a Booth encoding. Patterns which define Booth encoding usually stretch over more than one slice and we want to eliminate these nodes during “Global-Elimination” to reduce the size of the carries.

Before we apply “Global-Elimination” we locally eliminate variables in the sliced

Algorithm 11: Local-Elimination

Input : Ordered sliced Gröbner bases G_i , constant l
Output : Simplified ordered sliced Gröbner bases G_i

```

1  $p_0, \dots, p_m \leftarrow \text{Ordered-List-of-Polynomials}(G_i(C));$ 
2  $\tau \leftarrow 1;$ 
3 while  $\tau$  do
4    $\tau \leftarrow 0;$ 
5   for  $j \leftarrow 0$  to  $m$  do
6     if  $\text{Check-for-Elimination}(p_j)$  then
7        $q \leftarrow \text{Find-Parent-Polynomial}(p_0, \dots, p_{j-1});$ 
8        $q \leftarrow \text{D-reduction}(q, p_j, l);$ 
9        $G_i(C) \leftarrow G_i(C) \setminus \{p_j\};$ 
10       $\tau \leftarrow 1;$ 
11 return  $G_i(C)$ 

```

Algorithm 12: D-reduction in AMULET

Input : Two polynomials p and $q \in \mathbb{Z}[X]$, constant l
Output : Remainder r of D-reducing p modulo q

```

1  $p_d \leftarrow \text{Divide-by-lm}(p, q);$ 
2  $p_m \leftarrow \text{Multiply}(p_d, q, l);$ 
3  $r \leftarrow \text{Add}(p, p_m, l);$ 
4 return  $r$ 

```

Gröbner bases $G_i(C)$. We described in [62] a procedure which allows us to locally eliminate variables without violating the D-Gröbner basis property.

Theorem D.15 ([62]). *Let $P \subseteq \mathbb{Z}[X]$ be a D-Gröbner basis of $\langle P \rangle$ with UMLT. Let $q \in P$ be a polynomial with $\text{lt}(q) = z$ and no other polynomial $p \in P$ contains z . Then $P \setminus \{q\}$ is a D-Gröbner basis with UMLT for the ideal $J = I \cap \mathbb{Z}[X \setminus \{z\}]$.*

We use the conclusion of Thm.D.15 as follows. Assume $z \in X$ shall be eliminated and let $p, q \in G_i(C)$ be such that $\text{lt}(p) = z$ and z is contained in q . To eliminate z of $G_i(C)$, we D-reduce q by p and subsequently delete the polynomial p .

The pseudo-code for “Local-Elimination” is shown in Alg. 11. We iterate over the polynomials p_j in $G_i(C)$ and check whether the variable of the leading term is a candidate for local elimination, i.e., we check if the variable is contained in exactly one other polynomial of the same slice and if it is not marked as a carry variable. If both checks succeed, we search for the polynomial q , which contains the leading term of p_j and apply D-reduction of q by p_j . Since the polynomials are ordered, we only have to consider polynomials $p_i > p_j$ in this search.

Algorithm 12 shows how D-reduction is implemented in AMULET. In “Divide-by-lm” we use the UMLT property. Let $v = \text{lt}(q)$. We iterate over the monomials m in p and check whether v is contained in m . If v is contained in m we generate a

monomial m' which consists of all variables of m different from v . Furthermore we set $\text{coeff}(m') = \text{coeff}(m)$. All these generated monomials m' are summed up and define the polynomial p_d . The operations “Multiply” and “Add” correspond to the elementary polynomial operations. For addition we use the fact that polynomials are ordered lists of monomials. We iterate over the two polynomials simultaneously and merge them in an interleaved way. More precisely, we start at the leading monomials of p and p_m and compare them. If the monomials are different, we add the larger monomial to r . If the monomials are equal we generate a new monomial, which has the same term and the coefficient is the sum of the two coefficients. This way we ensure that r is again ordered. For multiplication we multiply each monomial of p_d with each monomial of q and sort the calculated monomials. In both operations we directly divide the calculated coefficients by the constant l in order to achieve reduction by l . We further handle reduction by $B(G_i)$ implicitly, i.e., we replace x^i by x during multiplication too, for all $i > 0$.

Example D.16. Let $p = -a + 2bc - bd$ and $q = -b + 2xy \in \mathbb{Z}_4[X]$. The intermediate results of Alg. 12 are $p_d = 2c - d$, $p_m = -2bc + bd - dxy$ and $r = -a - dxy$.

Let us continue the discussion of Alg. 11. The polynomial q is replaced by the remainder of the D-reduction step and the polynomial p_j is eliminated from the sliced Gröbner basis $G_i(C)$. We repeat variable elimination until no more polynomial in $G_i(C)$ can be considered for local elimination, i.e., all variables of its leading term are either carries or contained in multiple polynomials of the same slice. The rewritten Gröbner basis $G_i(C)$ is returned.

Now consider Alg. 10, where we repeat “Local-Elimination” for all sliced Gröbner bases $G_i(C)$. In “Global-Elimination” we eliminate the variables which were previously marked, independently how often they occur or whether they are carries. To this end we have to iterate over all polynomials in $G(C)$, finding their parent polynomials for D-reduction.

After variable elimination we apply the incremental checking algorithm as presented in Alg. 8. We start with s_{2n-1} and apply D-reduction by the polynomials in $G_{2n-1}(C)$. In order to consider each polynomial of a slice only once for D-reduction, we D-reduce by the polynomials in $G_i(C)$ in reverse topological order. After we applied D-reduction by all polynomials of a slice we add to the remainder C_i the partial products and output bit of the next smaller slice in order to derive $C_i + \alpha_{i-1}2^{i-1}s_{i-1} - 2^iP_{i-1}$. After reducing by $G_0(C)$, we check whether the final result is 0.

D.5 Proof Generation

Formal verification derives correctness of a given system. However, the process of verification as well as the implementation might not be bug-free. A common approach to increase the confidence in automated reasoning tools is to generate proof certificates, which are checked by independent proof checkers.

For example, providing certificates of unsatisfiability is mandatory in the SAT competition since 2013. Generating and checking proofs efficiently is a lively research

topic in the SAT community and several proof formats such as RUP [41], DRUP [49], DRAT [42] and LRAT [37] are available.

In order to provide proof certificates for reasoning tools using computer algebra we developed in [90] a proof format, called practical algebraic calculus (PAC), which is based on the polynomial calculus [34] and captures whether a polynomial is contained in the ideal generated by a given set of polynomials.

Our tool AMULET is able to generate proof certificates in the PAC format [90] to validate the result of Alg. 10. These proofs can be checked by our independent proof checker PACTRIM [90]. We write proofs as sequences, where each rule is of the following form:

$$\begin{array}{ll}
 d + : p_i, p_j, p_i + p_j; & \begin{array}{l} p_i, p_j \text{ appearing earlier in the proof} \\ \text{or are contained in } G(C) \cup \{l\} \\ \text{and } p_i + p_j \text{ being reduced by } B(X) \end{array} \\
 d * : p_i, q, qp_i; & \begin{array}{l} p_i \text{ appearing earlier in the proof or} \\ \text{or is contained in } G(C) \cup \{l\} \\ \text{and } q \in R[X] \text{ being arbitrary} \\ \text{and } qp_i \text{ being reduced by } B(X) \end{array}
 \end{array}$$

These rules model the properties of an ideal, as given in Def. D.4. Thus every conclusion polynomial $p = p_i + p_j$ or $p = qp_i$ is an element of $\langle G(C) \cup \{l\} \rangle$. We extend the proof rules by an optional deletion information d , similar to clause deletion in [49]. If d occurs in a proof rule the antecedents p_i and p_j are deleted from the inference set, which helps to reduce the memory usage of PACTRIM.

We do not explicitly write down proof rules when reducing a Boolean value constraint. Similar to verification, reduction by $B(X)$ is computed implicitly, e.g., $* : x, x, x;$ is a valid proof rule.

Definition D.17. The *length* of a PAC proof is defined as the number of generated proof rules. The *size* is determined as the total number of monomials in the conclusion polynomials, counted with repetition and *degree* defines the maximum degree seen in the conclusion polynomials.

PAC proofs are generated in AMULET as follows. The set of polynomials $G(C) \cup \{l\}$, which are defined in line 5 of Alg. 10 determines the initial constraint set. The specification \mathcal{L} defines the target polynomial, i.e., the polynomial which is checked whether it is inferred by the proof rules. Proof rules have to be generated whenever polynomials are manipulated, that is for variable elimination, either locally or globally and during incremental reduction in Alg. 10.

For variable elimination we produce proof rules which simulate D-reduction of a polynomial p by a polynomial q , cf. Alg. 12. Note that p and q are both contained in $G(C)$ and thus appear earlier in the proof. In general two rules are generated, a multiplication rule and an addition rule:

$$(d) * : q, p_d, p_m; \quad d + : p, p_m, r;$$

In AMULET reducing the polynomials p_m and r by the constant l is handled implicitly. However to generate a complete PAC proof, we need to generate explicit proof rules which model D-reduction of p_m and r by the constant l .

After a polynomial q was used for D-reduction during “Local-Elimination” we know, that we do not have to consider q anymore, as p was the only polynomial containing the leading variable of q . Thus we can delete q from the constraint set, which we indicate by the optional parameter “ d ”. For “Global-Elimination” we have to be more careful with deletion, as the polynomial q may be used multiple times for elimination. In both cases we eliminate p as we want to continue with the rewritten polynomial r .

For monitoring the incremental reduction we also have to generate proof rules which simulate D-reduction of p by q . However in contrast to variable elimination, p is not part of the constraint set and thus the addition rule would raise an error. On the other hand recall that all elements of an ideal can be represented as a linear combination of the generators of the ideal, cf. Def D.4. To simulate the linear combination we generate a multiplication PAC rule $(d) * : p_d, q, p_m$; for each D-reduction step and store the computed factor p_m . After finishing D-reduction of a slice S_i , we sum up all the generated factors p_m to derive the carry recurrence relations. After deriving all carry recurrence relations we sum them up and if the circuit is correct the final polynomial is the specification of the circuit. In both cases we sum up the polynomials in a tree-like approach, i.e., $((p_1 + p_2) + (p_3 + p_4))$ which is more beneficial compared to summing up the polynomials in order $((p_1 + p_2) + p_3) + p_4$ as this keeps the number of monomials in the intermediate summands smaller.

D.6 Proof Size

Proof complexity aims to analyze computational resources and allows us to reason about the performance of solvers. In this section we want to elaborate the efficiency of AMULET and investigate the complexity of the generated proofs. In particular we are interested in the proof length, proof size and degree.

Proof complexity for multiplier circuits is for example studied in [8], where it is shown that verifying ring properties, e.g., commutativity of multiplication, admit polynomial resolution proofs for simple multipliers. Motivated by this result we experimentally show in [90] that checking commutativity of simple multipliers generates PAC proofs of quadratic length and cubic size. However these proofs are generated using existing computer algebra systems [102].

In this section we investigate the complexity of the proofs generated by AMULET for specific family of multipliers, more precisely btor-multipliers, which implement multiplication of unsigned integers. These multipliers are generated by Boolector [83] and have a simple architecture as can be seen in Fig. D.5a for input bit-width 4. They are also used in the experiments of [90] and correspond to the array multipliers as defined in [8]. In contrast to [8, 90] we investigate the complexity for verifying the correctness of the circuit. For the proof length and degree we can give a precise bound while for proof size we derive an upper bound.

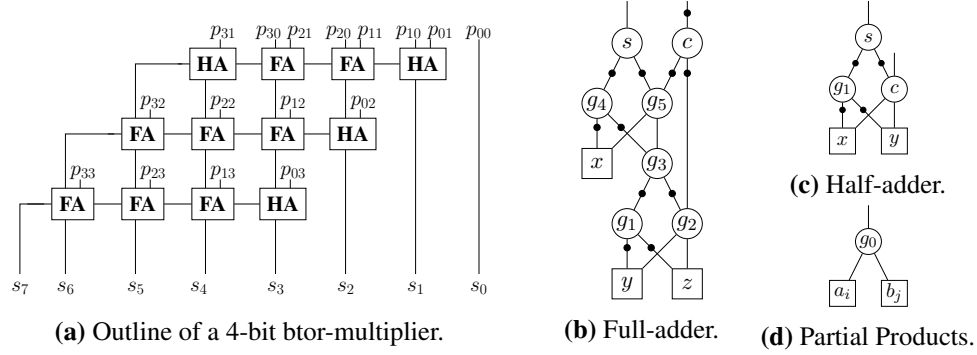


Figure D.5: The architecture of btor-multipliers and their representation as AIGs.

Lemma D.18. *Let C be a n -bit btor-multiplier. C contains n half-adders and $n^2 - 2n$ full-adders.*

Proof. As Fig. D.5a shows, we can clearly identify rows and columns in the btor-multipliers. Let Ab_i denote the sequence of all partial products $a_j b_i$ for $0 \leq j \leq n-1$. The first row of full- and half-adders (as seen from the circuit inputs) in C sum up the partial products Ab_0 and Ab_1 . In row k for $k \geq 2$ the partial products Ab_k are added to the sum-outputs of the adders of row $k-1$. Thus a btor-multiplier consists of $n-1$ rows.

In row k with $k \geq 2$ the adders sum up two bit-vectors of length n , which requires n adders. As we do not have an incoming carry the first adder is a half-adder and the remaining $n-1$ adders are full-adders. In the first row the partial product $a_0 b_0$ is directly processed to be output s_0 . Thus only $2n-1$ bits are summed up, which requires 2 half-adders and $n-2$ full-adders. Consequently btor-multipliers have $(n-2)(n-1) + n-2 = n^2 - 2n$ full-adder and $n-2+2 = n$ half-adders. \square

Lemma D.19. *Let C be a btor-multiplier of input bit-width n . The number of variables is $8n^2 - 7n$ and the size of $G(C)$ is $8n^2 - 9n$.*

Proof. The total number of variables consists of the input variables $a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1}$, output variables s_0, \dots, s_{2n-1} and the internal variables g_0, \dots, g_k . It is easy to see that we need $4n$ variables for the inputs and outputs. The internal nodes either represent partial products or they represent internal nodes of full- and half-adders, cf. Fig. D.5d, D.5b and D.5c. Generating a partial product needs one variable $p_{ij} = a_i b_j$, thus n^2 variables are needed to identify the partial products. According to Lemma D.18 btor-multipliers have n half-adders, each consisting of 3 nodes and $n^2 - 2n$ full-adders consisting of 7 nodes. Hence the total number of variables is $4n + n^2 + 3n + 7(n^2 - 2n) = 8n^2 - 7n$.

Each variable, despite of the $2n$ input variables, generates either a gate polynomial or a linking polynomial. Thus we have $8n^2 - 9n$ polynomials. \square

For proof length we measure the number of generated PAC rules. Because of the specific structure of btor-multipliers D-reduction by the constant $l = 2^{2n}$ is not necessary.

Thus each D-reduction step in variable elimination and in the incremental reduction algorithm produces at most two proof rules, namely one multiplication rule and one addition rule. Furthermore the partial products are generated using AND-gates, thus “Global-Elimination” is not necessary and proof rules are only generated in “Local-Elimination” and “Incremental-Reduction”. Hence each gate constraint in $G(C)$ is considered only once for D-reduction and thus we have an upper bound of $2(8n^2 - 9n) = 16n^2 - 18n$ proof rules. This bound is not tight as the following lemma shows.

Theorem D.20. *The proof length of n -bit btor-multipliers produced in AMULET is $16n^2 - 20n - 1$.*

Proof. In “Local-Elimination” all AIG nodes g_k , which occur in the linking polynomials $-s_i + g_k$ are eliminated. The variable g_k which links s_{2n-1} is not eliminated, as it acts as a carry. Since the coefficient of the variables g_k in the linking polynomials is 1, only the addition rule is required for D-reduction. We have $2n - 1$ such rules.

In the full- and half-adders the variables with only one parent get eliminated, that is g_1 and g_4 in Fig. D.5b and g_1 in Fig. D.5c. In total $2(n^2 - 2n) + n = 2n^2 - 3n$ variables are eliminated and each of these eliminations requires two rules. Hence “Local-Elimination” totally requires $2n - 1 + 2(2n^2 - 3n) = 4n^2 - 4n - 1$ proof rules.

For “Incremental-Reduction” we need to consider the multiplication rules as well as the summation rules. After variable elimination $8n^2 - 9n - (2n - 1) - (2n^2 - 3n) = 6n^2 - 8n + 1$ polynomials remain in $G(C)$. Each of them, except for the single polynomial $-s_0 + a_0b_0$ in S_0 produces a multiplication rule. Thus $6n^2 - 8n$ multiplication rules are generated.

The $6n^2 - 8n$ factors plus the polynomial $-s_0 + a_0b_0$, are summed up slice-wise to produce the carry recurrence relations. The sum of these carry recurrence relations produces the multiplier specification. Thus $6n^2 - 8n$ additions are necessary. Collecting all the generated proof rules leads to the final number of $4n^2 - 4n - 1 + 2(6n^2 - 8n) = 16n^2 - 20n - 1$ proof rules. \square

Theorem D.21. *The degree of the PAC proof of n -bit btor-multipliers is 3.*

Proof. The degree of the polynomials in the initial constraint set is at most 2, since the degree of the polynomials induced by AIG nodes is 2 and the linking polynomials have degree 1.

The degree of the PAC proof can only increase in multiplication rules. In the remainder of the proof we will heavily use the annotation of the variables as in Fig. D.5b and Fig. D.5c.

In “Local Elimination” we eliminate g_1 and g_4 from the full-adders and g_1 from the half-adders. As they have the same internal structure, we only discuss elimination of g_1 from half-adders. To eliminate g_1 , the polynomial $p = -s + (1 - c)(1 - g_1)$ is D-reduced by $q = -g_1 + (1 - x)(1 - y)$. Hence “Divide-by-lm” of Alg. 12 yields $p_d = c - 1$. Consequently, the resulting polynomial of multiplying q and p_d is $p_m = -g_1c + g_1 + xyc - xy - xc + x - yc + y + c - 1$ and has degree 3.

In the slicing algorithm btor-multipliers are partitioned in such a way, that all nodes of a full- and half-adder belong to the same slice. Thus the internal nodes of full- and

half-adders are reduced in sequence, which has the consequence, that summing up the factored gate polynomials of internal adder nodes yields the adder specifications $2(1 - c) + s = x + y + z$ for full-adders and $2c + s = x + y$ for half-adders. We use this observation to determine the degree of the factors.

We first discuss half-adders. After local elimination of g_1 half-adders are modeled by the polynomials $-s + (c - 1)(xy - x - y)$ and $-c + xy$. The following multiplication rules are generated during incremental reduction. The constant α depends on the slice in which the half-adder belongs. Both conclusion polynomials have degree 3 and adding them yields the specification of a half-adder.

$$\begin{aligned} * : & -s + (c - 1)(xy - x - y), \quad \alpha, \quad -\alpha s + \alpha cxy - \alpha cx - \alpha cy - \alpha xy + \alpha x + \alpha y; \\ * : & -c + xy, \quad \alpha(xy - x - y + 2), \quad -\alpha cxy + \alpha cx + \alpha cy - 2\alpha c + \alpha xy; \end{aligned}$$

For full-adders the following factors are generated. All of them have at most degree 3.

$$\begin{aligned} * : & -s + (g_5 - 1)(g_3x - g_3 - x), \quad \alpha, \\ & -\alpha s + \alpha g_5 g_3 x - \alpha g_5 x - \alpha g_5 g_3 - \alpha g_3 x + \alpha g_3 + \alpha x; \\ * : & -c + (1 - g_5)(1 - g_2), \quad -2\alpha, \\ & 2\alpha c - 2\alpha g_5 g_2 + 2\alpha g_5 + 2\alpha g_2 - 2\alpha; \\ * : & -g_5 + g_3x, \quad \alpha(g_3x - g_3 - 2g_2 - x + 2), \\ & -\alpha g_5 g_3 x + \alpha g_5 g_3 + 2\alpha g_5 g_2 + \alpha g_5 x - 2\alpha g_5 - 2\alpha g_3 g_2 x + \alpha g_3 x; \\ * : & -g_3 + (g_2 - 1)(yz - y - z), \quad \alpha(-2g_2x + 1), \\ & 2\alpha g_3 g_2 x - \alpha g_3 + \alpha g_2 yz - \alpha g_2 y - \alpha g_2 z - \alpha yz + \alpha y + \alpha z; \\ * : & -g_2 + yz, \quad \alpha(yz - y - z + 2), \\ & -\alpha g_2 yz + \alpha g_2 y + \alpha g_2 z - 2\alpha g_2 + \alpha yz; \end{aligned}$$

All polynomials, which model partial products are only multiplied by constants. Thus we never generated a polynomial which has a degree larger than 3. \square

In contrast to proof length and degree we are only able to determine an upper bound for proof size.

Theorem D.22. *The proof size of n -bit btor-multipliers is in $\mathcal{O}(n^2 \log(n))$.*

Proof. As in the previous proofs we distinguish between “Local Elimination” and “Incremental Reduction”. Eliminating g_k from the $2n$ linking polynomials $-s_i + g_k$ needs only one addition. The conclusion polynomial has at most 5 monomials, since each gate constraint contains at most 5 monomials.

Elimination of g_1 and g_4 in the full-adders and g_1 in the half-adders produces one multiplication rule and one addition rule. In the proof of Thm. D.21 we listed the conclusion polynomial p_m of the multiplication, which has size 10. Adding p_m to $-s + (1 - g_1)(1 - c)$ yields a polynomial with 7 monomials. Since we eliminate two variables from each full-adder and one variable from each half-adder, we eliminate $2n^2 - 3n$ variables. Each elimination produces 17 monomials. Thus “Local Elimination” produces at most $5(2n) + 17(2n^2 - 3n) = 34n^2 + 41n$ monomials.

In “Incremental Reduction” we need to consider the multiplication rules as well as the addition rules which add up the polynomials slice-wise and then totally to gain the word-level specification.

The n^2 polynomials defining the partial products are multiplied by constants 2^i , thus each conclusion polynomial has 2 monomials. We have already written down each multiplication rule for the full- and half-adders in the proof of Thm. D.21. Counting the monomials yields 32 monomials for each full-adder and 12 monomials for each half-adder. Thus in total $2n^2 + 32(n^2 - 2n) + 12n = 34n^2 - 52n$ monomials are needed in the multiplication rules.

After the factors are generated, they are added up in a tree-like approach, as discussed at the end of Sect. D.5. If m polynomials are added, the depth of the corresponding addition tree is $\lceil \log(m) \rceil + 1$.

First the polynomials within one slice are summed up. The biggest slice is S_{n-1} , which contains $n - 2$ full-adders, 1 half-adder and n partial products. Thus in total $6n - 8$ polynomials are added. For simplicity we drop the constant and assume $6n$ polynomials are added. The depth of the tree is $\lceil \log(6n) \rceil + 1 < \lceil \log(6) \rceil + \lceil \log(n) \rceil + 1 < \lceil \log(n) \rceil + 4$.

It can be seen in the proof of Thm. D.21, that each polynomial contains at most 8 monomials. Thus the initial layer of the addition tree has at most $48n$ monomials. Let us assume adding two polynomials does not cancel any monomials. Thus in layer i of the addition tree, the polynomials have $2^i \cdot 8$ monomials. Since each layer has $\frac{1}{2^i}(6n)$ polynomials, the total number of monomials for each layer is $48n$. Adding up one slice produces at most $48n(\lceil \log(n) \rceil + 4) = 48n\lceil \log(n) \rceil + 192n$ monomials. Since we have $2n$ slices, we have at most $96n^2\lceil \log(n) \rceil + 384n^2$ monomials.

We add up these carry recurrence relations to gain the word-level specification. We have $2n$ carry recurrence relations and each of them contains one monomial for the output variable s_i , at most $n - 1$ monomials for the incoming carries and $n - 1$ monomials for the outgoing carries and at most n partial products and one constant monomial. Adding two consecutive carry recurrence relations cancels the matching outgoing and incoming carries. Thus after adding two initial polynomials, the resulting polynomials contains 2 monomials for the output bits, at most $2n - 2$ monomials related to carries and at most $2n$ partial products and a constant. Let $m = \lceil \log(2n) \rceil \leq \lceil \log(n) \rceil + 1$. We have $m + 1$ addition layers and each layer contains $\frac{2n}{2^i}$ polynomials. Thus the upper bound of monomials is

$$\begin{aligned} \sum_{i=0}^m \frac{2n}{2^i} & \left(\overbrace{2^i}^{\text{output}} + \overbrace{n-1}^{\text{carry in}} + \overbrace{n-1}^{\text{carry out}} + \overbrace{2^i n}^{\text{p.products}} + \overbrace{1}^{\text{constant}} \right) = \\ & 2n \sum_{i=0}^m (n+1) + 2n(2n-1) \sum_{i=0}^m \frac{1}{2^i} < \\ & 2n(n+1)(\lceil \log(n) \rceil + 2) + 4n^2(2 - \frac{1}{4n}) = \\ & 2n^2\lceil \log(n) \rceil + 2n\lceil \log(n) \rceil + 12n^2 + 3n. \end{aligned}$$

Altogether our upper bound yields the polynomial $98n^2\lceil \log(n) \rceil + 2n\lceil \log(n) \rceil + 464n^2 - 8n$ and thus we are in $\mathcal{O}(n^2 \log(n))$. \square

We could clearly improve this bound, as we now considered all $2n$ slices to contain the same number of polynomials as the largest slice. Furthermore monomials do cancel when the polynomials within a slice are summed up. The real proof size as well as the estimated upper bound can be seen in Fig. D.6. We further added the function $50n^2 \log(n)$ in the plots, which also seems to be sufficient as an upper bound. In Fig. D.7 we show the relative errors of the upper bounds.

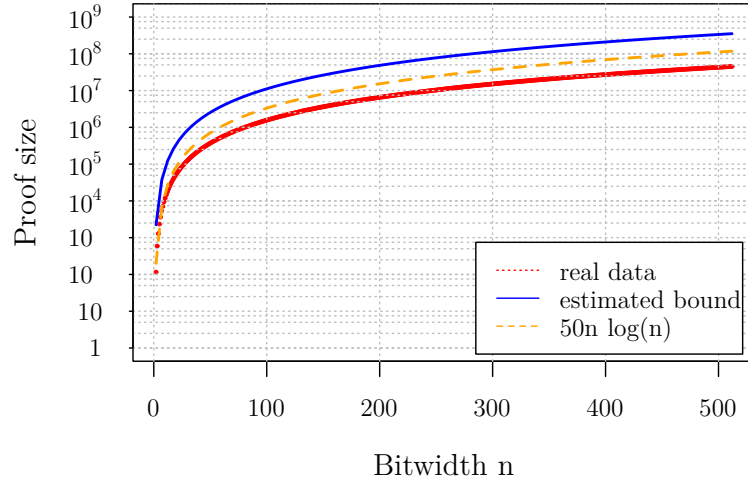


Figure D.6: Proof size for $n = [2, 512]$.

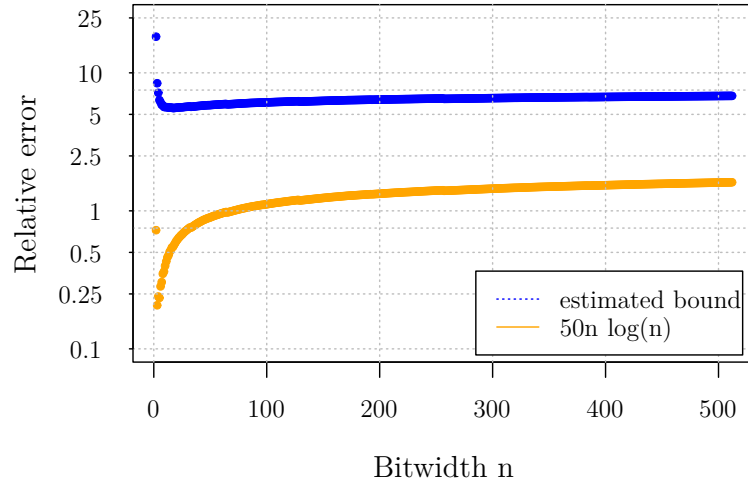


Figure D.7: Relative errors of upper bounds for $n = [2, 512]$.

D.7 Conclusion

In this paper we presented our tool **AMULET**, a state-of-the-art tool to automatically verify and certify the correctness of large gate-level integer multipliers. We gave an introduction into the problem of arithmetic circuit verification and discussed our state-of-the-art solving method which combines SAT and computer algebra. Certain parts, more precisely complex final stage adders, of the multiplier are detected and replaced by simple ripple-carry adders. The correctness of the replacement is checked by SAT solvers and the rewritten multiplier is verified using computer algebra. We presented details of the underlying algorithms to detect final stage adders and rewrite the multipliers, originally introduced in [62]. Furthermore we reconsidered our incremental verification algorithm and discussed the procedure of generating proof certificates. For one specific simple type of multipliers we showed that we are able to generate proof certificates with length in $\mathcal{O}(n^2)$ and size in $\mathcal{O}(n^2 \log(n))$. In the future we want to be able to extend our methods to synthesized multipliers where technology mapping is applied. Investigating floating points and other word-level operators is interesting future work too.



Paper E

From DRUP to PAC and Back

To be published In Proceedings of the Design, Automation & Test in Europe Conference (DATE 2020), 4 pages, Grenoble, France, 2020.

This paper was submitted as a regular paper consisting of 6 pages, and was accepted as a short paper, which is restricted to 4 pages. This thesis contains an extended version of the accepted paper, which is closer to the submitted version.

Authors Daniela Kaufmann, Armin Biere and Manuel Kauers.

Acknowledgement This work is supported by the Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), P31571-N32, SFB F5004, LIT AI Lab funded by the state of Upper Austria.

Differences In contrast to the conference paper, this paper contains more experimental data and is extended by examples. Furthermore, a more sophisticated introduction of SAT and the DRUP format is given in Sect. E.2.2.

Fixes Changed the equation $-f_i + l_i - 1 = 0$ to $-f_i - l_i + 1 = 0$. Added link to experimental data.

Abstract Currently the most efficient automatic approach to verify gate-level multipliers combines SAT solving and computer algebra. In order to increase confidence in the verification, proof certificates are generated. However, due to different solving techniques, these certificates require two different proof formats, namely DRUP and PAC. A combined proof has so far been missing. Correctness of this approach can thus only be trusted up to the correctness of compositional reasoning. In this paper we show how to generate a single proof in one proof format, which then allows us to certify correctness using one simple proof checker. We further investigate empirically the effect on proof generation and checking time as well as on proof size. It turns out that PAC proofs are much more compact and faster to check.

E.1 Introduction

Fully automated verification of gate-level multiplier circuits is still considered to be hard. The currently most effective approach relies on computer algebra [31, 62, 80]. Whereas the authors of [31, 80] employ only algebraic reasoning, we further combine *Boolean satisfiability* (SAT) solving [62]. We conjectured in [62], that certain final stage adders are a real challenge for the computer algebra approach. On the other hand, these adders can easily be verified using SAT solvers. In our approach we are replacing complex adders by simple ripple carry adders (RCA). The correctness of the substitution is proved by SAT solvers and the rewritten multiplier is verified using the computer algebra approach.

We increase the trust in the verification result by generating certificates in [62], which can be checked by independent proof checkers. Since our technique relies on two different reasoning techniques, also two proof certificates in different proof formats are produced. The polynomial reduction algorithm produces an algebraic proof in the *practical algebraic calculus* (PAC) [90] and SAT solvers produce clausal proofs in the *delete reverse unit propagation* (DRUP) proof format [47]. These proofs are checked by two different proof checkers, leaving a hole in the certification argument. Compositional reasoning using interactive theorem proving [54] could close this gap but is not fully automatic.

In this work we present how these two proof formats used in [62] can be merged into one common proof format. Although this paper is tailored to the use case of [62], the proposed methods are not limited to this particular application.

We are able to convert a DRUP proof into a PAC proof. On the other hand, our results for converting a PAC proof into a DRUP proof can be considered to provide a lower bound on the proof size. In the conversion we use a *satisfiability modulo theories* (SMT) encoding and thus are not able to track any rewriting employed by SMT solvers as a DRUP proof.

Our experiments generate proofs in a single proof format. It turns out that PAC proofs are superior to DRUP proofs, as DRUP proofs are around three orders of magnitude larger than PAC proofs. Additionally, as already mentioned, our DRUP proofs do not yet cover all necessary proof steps.

E.2 Preliminaries

We recapitulate the two proof formats DRUP and PAC and further summarize the state-of-the-art [62] for automatic verification of unsynthesized multiplier circuits.

E.2.1 Algebra and the PAC format

In this section we introduce basic concepts of algebra [35] and describe the PAC proof format [90].

A nonempty subset of polynomials $I \subseteq \mathbb{Z}[X]$ is called an *ideal* if $\forall p, q \in I : p+q \in I$ and $\forall p \in \mathbb{Z}[X] \forall q \in I : pq \in I$. A set $P = \{p_1, \dots, p_s\} \subseteq \mathbb{Z}[X]$ is called a *basis* of

I if $I = \{p_1q_1 + \dots + p_sq_s \mid q_1, \dots, q_s \in \mathbb{Z}[X]\}$. We then say I is generated by P and write $I = \langle P \rangle$.

Let $f \in \mathbb{Z}[X]$ and $P \subseteq \mathbb{Z}[X]$. We are interested whether the polynomial equation $f = 0$ is implied by the equations $p = 0$ with $p \in P$. This question is also called ideal membership problem: Given f and P as above decide whether $f \in \langle P \rangle$.

We focus on gate-level circuit verification, where all variables $x \in X$ represent logic gates and thus take only values in $\{0, 1\}$. This is enforced by *Boolean value constraints* of the form $x(1 - x) = 0$. Let $B(X) = \{x(1 - x) \mid x \in X\} \subseteq \mathbb{Z}[X]$ be the set of Boolean value constraints for X . Each gate of the circuit is encoded by a polynomial relation, called *gate polynomial*, which are collected in P . Consequently the ideal membership problem we actually want to solve is formulated as: Given $f \in \mathbb{Z}[X]$ and $P \subseteq \mathbb{Z}[X]$, decide whether $f \in \langle P \cup B(X) \rangle$.

The practical algebraic calculus (PAC) format allows us to capture the derivation of an equation $f = 0$ from a given set of polynomial equations P and thus $f \in \langle P \cup B(X) \rangle$.

Proofs are sequences of proof rules, which model the ideal properties, where each rule has the following form:

$$\begin{array}{ll}
 + : p_i, p_j, p_i + p_j; & \begin{array}{l} p_i, p_j \text{ appearing earlier in the proof} \\ \text{or are contained in } P \\ \text{and } p_i + p_j \text{ being reduced by } B(X) \end{array} \\
 * : p_i, q, qp_i; & \begin{array}{l} p_i \text{ appearing earlier in proof or in } P \\ \text{and } q \in \mathbb{Z}[X] \text{ being arbitrary} \\ \text{and } qp_i \text{ being reduced by } B(X) \end{array}
 \end{array}$$

As described in [62] “being reduced by $B(X)$ ” means, that each occurrence of x_i^d with $d > 1$ is immediately replaced by x_i , e.g., $x * x = x$, thus $* : x, x, x;$ is a valid proof rule.

Example E.1. Let $P = \{-x + 3z, 2xz\} \subseteq \mathbb{Z}[x, y, z]$ and let $f = -2x \in \mathbb{Z}[x, y, z]$. The proof shows $f \in \langle P \cup B(X) \rangle$:

$$\begin{array}{ll}
 * : & -x+3z, \quad 2x, \quad -2x+6xz; \\
 * : & \quad 2xz, \quad -3, \quad -6xz; \\
 + : & -2x+6xz, \quad -6xz, \quad -2x;
 \end{array}$$

E.2.2 SAT and the DRUP format

We briefly introduce the SAT problem and its common proof formats, following [47].

- A *literal* l is either a positive Boolean variable x or its negation \bar{x} .
- A *clause* C is a finite disjunction of literals. If a clause contains only one literal, we call it a *unit clause*.
- 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 $v \in \{\mathbf{true}, \mathbf{false}\}$, such that $\tau(x) = v \Leftrightarrow \tau(\bar{x}) = \neg v$, where $\neg \mathbf{true} = \mathbf{false}$ and $\neg \mathbf{false} = \mathbf{true}$.

The SAT problem seeks for an assignment such that a formula F evaluates to **true**. A formula evaluates to **true** if and only if every clause in the formula evaluates to **true**. A clause C evaluates to **true** if there exists $l \in C$ with $\tau(l) = \mathbf{true}$. If this is the case, we say the formula is *satisfiable*. If no satisfying assignment can be found, it is *unsatisfiable*.

A clause C is *redundant* w.r.t. a formula F , if $F \wedge C$ is satisfiable iff F is satisfiable. Redundant clauses are for example derived using *resolution* [92]: Given two clauses $C_1 = (a \vee x_0 \vee \dots \vee x_m)$ and $C_2 = (\bar{a} \vee y_0 \vee \dots \vee y_n)$, the clause $C = (x_0 \vee \dots \vee x_m \vee y_0 \vee \dots \vee y_n)$ can be resolved.

A further technique used in SAT solvers is called *unit propagation*: If a formula F contains a unit clause $C = l$, remove all clauses containing l and all occurrences of \bar{l} .

If a formula is satisfiable a satisfying assignment can act as witness. However if the formula is unsatisfiable more involved reasoning is required to derive proofs of unsatisfiability, also called *refutation*. Standard refutation proof formats are either resolution proofs or clausal proofs. Clausal proofs are easier to generate and are more compact than resolution proofs.

The most basic clausal proof format is *reverse unit propagation* (RUP) [41]. Let \bar{C} denote the negation of a clause C . If for example $C = a \vee b \vee \bar{x}$ then $\bar{C} = \bar{a} \wedge \bar{b} \wedge x$. We say C is a *RUP clause* if $F \wedge \bar{C}$ evaluates to **false**. A RUP proof is a sequence of RUP clauses containing the empty clause. A *delete reverse unit propagation* (DRUP) [49] proof extends RUP by adding deletion information to decrease the cost of proof validation [99]. DRUP can further be extended to the *deletion resolution asymmetric tautology* (DRAT) [42] format, which extends DRUP by allowing introduction of new variables.

Clausal DRUP proofs are checked through unit propagation. As a side effect a resolution proof [47] can be produced. The *TraceCheck* format is a common proof format for resolution proofs. It has the format “idx clause 0 antecedents 0”, where the antecedents are the indices of the clauses used in the resolution. Lines with trailing double zeros mark initial clauses.

Example E.2. This is an unsatisfiable CNF in DIMACS format (left) with DRUP (middle) and TraceCheck (right) proofs.

p cnf 3 5	-2 0	1 1 -2 -3 0 0
1 -2 -3 0	d 3 0	2 1 2 0 0
1 2 0	d 1 -2 -3 0	3 -1 -2 0 0
-1 -2 0	d -1 -2 0	4 -1 2 0 0
-1 2 0	0	5 3 0 0
3 0		6 -2 0 3 1 5 0
		7 0 4 2 6 0

E.2.3 State-of-the-art Circuit Verification

Multipliers are usually made up of three stages: (i) generation of partial products (PPG), (ii) accumulation of the partial products (PPA) and (iii) a final stage adder (FSA).

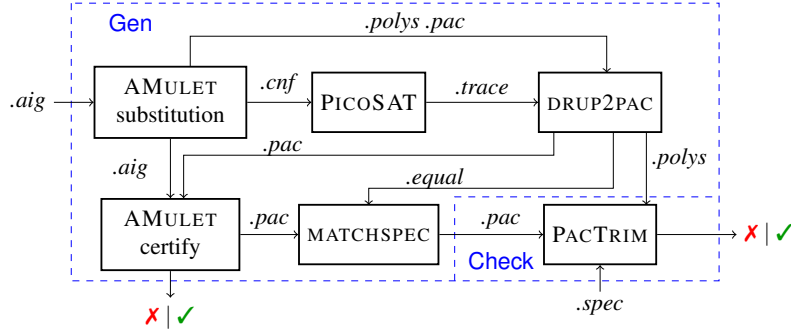


Figure E.1: Converting a DRUP proof into a PAC proof.

According to the state-of-the-art [62] the first two stages are easy for computer algebra, but some final stage adders, more precisely generate-and-propagate adders, are challenging for the computer algebra approach. However these adders are very easy for SAT solvers. In our technique of [62] we are given multipliers as And-Inverter-Graphs (AIG). We identify whether the final stage adder is a generate-and-propagate adder and if necessary substitute it with a simple RCA.

To verify that the original FSA is substituted with an equivalent RCA, a bit-level miter in CNF is generated, and checked by a SAT solver, which also produces a DRUP proof for certification. Correctness of the rewritten circuit is shown using computer algebra. For details see [31, 62, 80]. A PAC proof is computed alongside the polynomial reduction. In the toolflow of [62] the PAC proof is split into the “.polys” and “.pac” files, where “.polys” contains the initial set of polynomials P and “.pac” contains the PAC proof rules.

E.3 From DRUP to PAC

The necessary steps to merge the DRUP and the PAC proof of [62] into one single PAC proof are shown Fig. E.1.

Converting the DRUP proof into a PAC proof needs algebraic reasoning over the CNF encoding derived during adder substitution. As only the gate polynomials are contained in the constraint set we need to deduce the CNF encoding in PAC. Figure E.2 shows an AIG node and the corresponding encodings as propositional formulas and polynomial equations. Since in a satisfiable CNF every clause needs to evaluate to **true**, the CNF can be split into a system of “clausal equations” (on the right) encoding this property. We derive the corresponding system of polynomial equations from the initial polynomial relation by simple polynomial operations.

Example E.3. Using the fact that $x^2 - x = 0$, $b^2 - b = 0$ and $a^2 - a = 0$ we multiply the

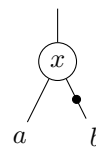
	Propositional Formula	Polynomial Relation
	$(x \leftrightarrow a \wedge \bar{b}) = \top$	$-x + a(1 - b) = 0$
CNF	$(x \vee \bar{a} \vee b) = \top$	$(1 - x)a(1 - b) = 0$
	$(\bar{x} \vee a) = \top$	$x(1 - a) = 0$
	$(\bar{x} \vee \bar{b}) = \top$	$xb = 0$

Figure E.2: Different encodings of the AIG node $x = a \wedge \bar{b}$.

polynomial equation $-x + a(1 - b)$ by different factors to derive the desired polynomials.

$$\begin{aligned}
 0 &= (-x + a(1 - b))(-ba + a) = (1 - x)a(1 - b) \\
 0 &= (-x + a(1 - b))(b - 1) = x(1 - a) \\
 0 &= (-x + a(1 - b))(-a) = xb
 \end{aligned}$$

We added to the original tool AMULET of [62] the ability to derive such polynomial encodings of CNFs during adder substitution.

The generated CNF miter of the adder substitution is given to the SAT solver PICOSAT [11]. We do not use CADICAL [15] as [62], because PICOSAT allows us to directly generate a resolution proof in the TRACECHECK format. The TRACECHECK proof alongside with the original CNF is passed on to our tool DRUP2PAC. In DRUP2PAC we encode the resolution proof as a PAC proof, by re-enacting the resolution steps in the given traces using algebraic reasoning. The following example shows the encoding of one resolution step.

Example E.4. Consider TraceCheck proof of Ex. E.2. Let $a = 1$, $b = 2$ and $c = 3$. We encode the first resolution step of rule 6 (resolving clause 3 and 1). Thus from $a \vee \bar{b} \vee \bar{c}$ and $\bar{a} \vee \bar{b}$ we resolve the clause $\bar{b} \vee \bar{c}$. The corresponding PAC encoding is:

$$\begin{aligned}
 * &: \quad \quad \quad b*a, \quad \quad c, \quad \quad c*b*a; \\
 + &: \quad -c*b*a+c*b, \quad c*b*a, \quad \quad c*b;
 \end{aligned}$$

However we do not want to derive the empty clause, as this corresponds to deriving the constant polynomial 1. Hence whenever we encounter the unit clause encoding the assumption of the miter in a trace, we remove it from the trace.

As a further optimization we internally apply *bit-flipping*, as for instance proposed in [94], on the algebraic level to keep the size of the intermediate polynomials small. It can be seen in the polynomial encoding of the CNF in Fig. E.2, that each positive literal l in a clause introduces a factor $(1 - l)$ in the corresponding polynomial encoding of the clause. As the PAC format uses only the expanded form of polynomials, expanding clauses with multiple positive literals leads to a tremendous growth in the polynomial encoding. In order to overcome this issue, we introduce for each literal l_i a bit-flipping polynomial $-f_i - l_i + 1 = 0$ in the constraint set and internally flip variables in the CNF such that only negative literals are contained. We monitor the bit-flipping in the clausal polynomials by generating corresponding PAC rules and add the bit-flipping polynomials to the constraint set.

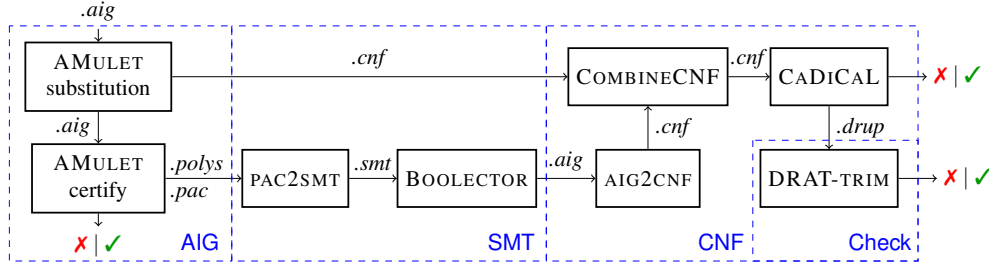


Figure E.3: Converting a PAC proof into a DRUP proof.

After translating the full TRACECHECK proof we derive for each pair of miter inputs the equations $s_i(s'_i - 1) = 0$ and $s'_i(s_i - 1) = 0$ using unit propagation. Encoding unit propagation in PAC is very similar to encoding resolution. Subtracting these polynomials leads to $s_i - s'_i = 0$ for each pair of input bits. We report these pairs in “equal”.

We certify the rewritten AIG using AMULET. At this point the specification which we derived in the PAC proof uses the outputs s'_i of the RCA. Our final tool MATCHSPEC generates PAC rules which replace every occurrence of s'_i by the corresponding bit s_i using the equations $s_i - s'_i = 0$. As a last step the generated proof and the original specification (in terms of s_i) is checked using PACTRIM.

E.4 From PAC to DRUP

We have seen how to encode a DRUP proof into PAC. However, not only is PAC more complex than DRUP (and DRAT), but PAC neither has certified proof checkers, while DRAT and thus DRUP can be translated to LRAT, for which such checkers exist [37]. Therefore it is natural to ask, whether it is possible to translate PAC proofs into DRUP.

In this section we give a positive but impractical answer. The first hurdle is to encode the specification into CNF. This can in principle be achieved using SMT over the theory of bit-vectors for a large enough bit-width followed by bit-blasting. However, at this point, we are not able to track rewriting within SMT solvers, which leaves a gap in the proof. A further issue of our encoding is that we only translate each PAC rule individually to SMT and CNF. We do not include a check that the specification of the circuit is derived at the end, which is another gap in our proof. Thus our resulting DRUP proof is far from being a complete proof, in the sense of covering every rewriting step. The size of these proofs can only be considered as an empirically derived lower bound.

Note that our translation introduces new variables and thus technically needs extended resolution (ER), thus actually DRAT. But we continue to use DRUP instead of DRAT to describe our approach. Our tool flow can be seen in Fig E.3. We apply adder substitution and certify the rewritten multiplier as in [62]. In our tool PAC2SMT we abstract the polynomial proof to a bit-vector proof. To this end we encode the PAC proof as an SMT problem over the theory of quantifier-free fixed size bit-vectors. Note that each variable in the PAC proof represents the input or output of a gate. As a consequence we encode

each variable in the PAC proof as a single bit and the coefficients are encoded as bit vectors. The length of the bit-vectors depends on the highest coefficient in the PAC proof.

Example E.5. Consider the following PAC rule

$$+ : 3x - z, 2y - 3x, 2y - z;$$

Checking the correctness of this rule can be encoded as:

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 1))
(declare-fun y () (_ BitVec 1))
(declare-fun z () (_ BitVec 1))
(assert
  (let (($v0 (bvadd (bvand #b011 ((_ sign_extend 2) x))
                    (bvand #b111 ((_ sign_extend 2) z)))))
    (let (($w0 (bvadd (bvand #b010 ((_ sign_extend 2) y))
                    (bvand #b101 ((_ sign_extend 2) x)))))
      (let (($p0 (bvadd (bvand #b010 ((_ sign_extend 2) y))
                    (bvand #b111 ((_ sign_extend 2) z)))))
        (let (($e0 (= (bvadd $v0 $w0) $p0)))
          (not $e0))))))
(check-sat)
```

In our encoding of the PAC rules we include the following optimization when single bits are multiplied with bit-vectors.

$$\begin{aligned} &(\text{bvand } \#b011 \text{ } ((_ \text{sign_extend } 2) \text{ } x)) = \\ &(\text{bvmul } \#b011 \text{ } ((_ \text{zero_extend } 2) \text{ } x)). \end{aligned}$$

We encode each rule of the PAC proof as a bit-vector equation and assume that the conjunction of all these equations is unsatisfiable. The SMT encoding is given to BOOLECTOR [83], which additionally is able to generate AIGs from bit-vector formulas. As discussed above, BOOLECTOR applies rewriting steps, which are not covered in the DRUP proof. Using the tool AIG2CNF from the Aiger library [16] we translate the AIG into a CNF. Nodes from AIGs can easily be encoded in CNF, as indicated in Fig. E.2.

At this point we have two CNF encodings. The first CNF is directly produced by AMULET and encodes the bit-level miter proving the correctness of the adder substitution. The second CNF encodes the translated PAC proof. Both CNFs are encoded to deliver a refutation, i.e., for a correct multiplier both CNFs should be unsatisfiable. More precisely each CNF encodes a miter, thus both CNF contain one unit clause $C_i = l_i$ which represents the assumption for the miter output.

The CNFs are merged by collecting all clauses, except the clause encoding the output assumption. The two output clauses $C_0 = l_0, C_1 = l_1$ are merged into the clause $l_0 \vee l_1$, thus either l_0 or l_1 needs to be true to satisfy the CNF. As we expect that both l_0 and l_1 are false, the clause $l_0 \vee l_1$ should be unsatisfiable, and thus the whole CNF too. The merged CNF is solved using the SAT solver CADICAL [15], which is instructed to generate a DRUP proof. Finally this proof is checked using DRAT-TRIM [99].

Table E.1: Proof Generation and Checking.

architecture	n	[62]																	
		DRUP			PAC			total	PAC				DRUP						
		gen	chk	size	gen	chk	size		gen	chk	total	size	aig	smt	cnf	check	total	size	
btor	8	-	-	-	0	0	1 181	0	-	-	-	-	0	1	7	4	12	831 546	
sp-ar-cl	8	0	0	471	0	0	1 746	0	0	0	0	37 167	0	1	28	14	43	2 211 172	
sp-bd-ks	8	0	0	504	0	0	1 846	0	0	0	1	57 079	0	1	23	12	36	1 964 878	
sp-dt-lf	8	0	0	515	0	0	1 675	0	0	0	1	53 850	0	1	20	11	32	1 842 288	
bp-ct-bk	8	0	0	413	0	0	1 976	0	0	0	1	46 115	0	1	185	155	340	4 420 593	
bp-wt-cl	8	0	0	759	0	0	2 092	0	0	0	1	67 951	0	1	143	137	282	4 317 440	
btor	16	-	-	-	0	0	5 181	0	-	-	-	-	0	3	136	177	316	11 079 431	
sp-ar-cl	16	0	0	1 299	0	0	7 962	0	2	2	3	185 588	0	7	300	264	570	19 317 884	
sp-bd-ks	16	0	0	2 140	0	0	8 356	0	2	2	4	209 249	0	7	283	290	579	17 989 961	
sp-dt-lf	16	0	0	1 167	0	0	7 787	0	1	1	2	136 349	0	6	279	277	562	18 153 668	
bp-ct-bk	16	0	0	1 029	0	0	7 205	0	1	1	2	128 720	0	7	TO	-	-	-	
bp-wt-cl	16	0	0	2 902	0	0	7 946	0	30	11	41	614 742	0	7	TO	-	-	-	
btor	32	-	-	-	0	0	21 629	0	-	-	-	-	0	32	2 887	TO	-	-	
sp-ar-cl	32	0	0	14 927	0	1	33 834	1	133	31	164	1 597 897	0	56	TO	-	-	-	
sp-bd-ks	32	0	0	17 528	0	1	34 958	1	20	8	28	817 956	0	54	TO	-	-	-	
sp-dt-lf	32	0	0	3 138	0	1	33 451	1	2	3	5	321 720	0	52	TO	-	-	-	
bp-ct-bk	32	0	0	2 276	0	1	27 312	1	1	2	3	217 128	0	49	TO	-	-	-	
bp-wt-cl	32	1	1	46 502	0	1	30 561	2	3 133	242	3 375	5 536 176	0	55	TO	-	-	-	
btor	64	-	-	-	2	2	88 317	4	-	-	-	-	2	410	TO	-	-	-	
sp-ar-cl	64	2	1	65 317	2	3	139 338	8	TO	-	-	-	2	577	TO	-	-	-	
sp-bd-ks	64	1	0	44 921	2	3	142 138	6	56	18	74	1 440 943	2	586	TO	-	-	-	
sp-dt-lf	64	0	0	28 772	2	3	138 539	6	10	10	19	816 572	3	561	TO	-	-	-	
bp-ct-bk	64	0	0	19 891	2	3	105 579	5	8	7	15	459 262	2	423	TO	-	-	-	
bp-wt-cl	64	8	6	42 199	2	3	118 573	19	TO	-	-	-	2	515	TO	-	-	-	

PPG: simple (sp), Booth (bp)

PPA: array (ar), Dadda tree (dt), compressor tree (ct), Wallace tree (wt)

TO = 3600 sec

FSA: carry look-ahead (cl), Ladner-Fischer (lf), Brent-Kung (bk)

Benchmarks are generated by the Arithmetic Module Generator [53].

E.5 Experiments

Our experiments were conducted on Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (with turbo-mode disabled). The time in Table E.1 is listed in rounded seconds (wall-clock time) and we measure the time from starting the tools until they finished or an error occurred, e.g., the time limit was reached, set to 3600 sec (1h), or the memory limit of 128 GB. Our experimental data, source code and benchmarks are available at <http://fmv.jku.at/drup2pac/>.

We consider various multiplier architectures used in our experiments in [62]. Benchmarks are generated with the Arithmetic Module Generator [53] and BOOLECTOR [83]. Except for the “btor” benchmarks from BOOLECTOR, the selected architectures contain generate-and-propagate final stage adders, thus adder substitution was required and applied as in [62] and hence a DRUP as well as a PAC proof were generated. These proofs are translated as explained in Sect. E.3 and Sect. E.4.

The first block shows the time for generating and checking the proofs as in [62]. For “DRUP” and “PAC” we present the time it takes to generate the corresponding proof and the time to check proofs using DRAT-TRIM [99] and PACTRIM [90]. Additionally we depict the sizes of the proofs. The proof size of DRUP proofs is measured by the number of added RUP clauses [47]. The size of PAC proofs is defined by the number of applied PAC rules [90].

The second block “PAC” shows the time for generating a single PAC proof as described in Sect. E.3. Almost all of the time in proof generation is used by converting the DRUP proof to a PAC proof, e.g., for “bp-wt-cl-32” our tool DRUP2PAC needs 3130 seconds. We are able to generate and check PAC proofs up to bit-width 32. The growth in the proof size depends highly on the benchmark, more precisely it depends on the generated DRUP proof. For instance for 32 bit, the increment of the PAC proofs is between factor 10 and factor 1600.

The third block “DRUP” lists the time for generating and checking a single DRUP proof as described in Sect. E.4. Column “aig” shows the time needed for adder substitution and generating the PAC proof. In column “smt” we present the time needed to generate an SMT proof as well as the time BOOLECTOR [83] needs to generate a CNF out of the SMT proof. The following column “cnf” lists the time we need to combine and solve the CNFs using CADICAL. We are only able to generate and check DRUP proofs up to 16 bit. The size of the DRUP proofs compared to the single PAC proofs increases drastically. Especially for the “btor” benchmarks, where no initial DRUP proof is generated, converting the PAC proof to the DRUP proof increases the size by three orders of magnitude. These are still not complete DRUP proofs. Neither rewriting, nor the extensions to encode bit-blasting (both requiring DRAT) are accounted yet.

E.6 Conclusion

State-of-the-art verification techniques of arithmetic circuits rely on SAT as well as computer algebra. However they lack a proof certificate in a single proof format. With

two proof formats we argue that additional manual compositional reasoning would be required to certify the verification. In this paper we present how to translate the clausal reasoning proof format DRUP into the algebraic proof format PAC and vice versa in order to produce one single proof certificate.

Translating DRUP proofs to PAC proofs requires algebraic reasoning. We include bit-flipping techniques in order to reduce the size of polynomials. As a further optimization we use the TRACECHECK format as input format, in order to directly determine the necessary polynomial equations.

To obtain DRUP from PAC proofs we encode the PAC proofs as an SMT problem, which then is translated into CNF using bit-blasting by an SMT solver. However, this intermediate step leaves gaps in the proof, since we are not able to track internals of SMT solving. Even though far from being complete proofs, they serve as empirically derived lower bounds on such clausal proofs. These proofs are three orders of magnitude larger than the corresponding PAC proofs.

As future work we want to be able to close the gap in generating DRUP proofs. Generating smaller proofs by applying more sophisticated reasoning is interesting as well.

Paper F

The Proof Checkers Pacheck and Pastèque for the Practical Algebraic Calculus

Submitted as a system description.

Authors Daniela Kaufmann, Mathias Fleury and Armin Biere.

Acknowledgement This work is supported by the Austrian Science Fund (FWF), NFN S11408-N23 (RiSE) and LIT AI Lab funded by the state of Upper Austria.

Abstract Generating and checking proof certificates is important to increase the trust in automated reasoning tools. In recent years formal verification using computer algebra became more important and is heavily used in circuit verification. An existing proof format which covers algebraic reasoning is the practical algebraic calculus. In this paper we present two independent proof checkers PACHECK and PASTÈQUE. The checker PACHECK checks algebraic proofs more efficiently than PASTÈQUE, but the latter is formally verified using the proof assistant Isabelle/HOL. Furthermore, we introduce extension rules to simulate essential rewriting techniques required in practice. For efficiency we also make use of indices for existing polynomials and include deletion rules too.

F.1 Introduction

Formal verification aims to guarantee the correctness of a given system with respect to a certain specification. However, the verification process might not be error-free. In order to increase the trust in verification results, it is a common approach to generate simple proof certificates, which can be checked by a stand-alone proof checker. For example, in the SAT competition certificates of unsatisfiability are required since 2013

and different resolution and clausal proof formats, such as DRUP, DRAT, and LRAT, are available [36, 41, 42, 47, 49].

Automated reasoning techniques based on computer algebra [21, 50, 58, 59, 60] provide the state of the art in verifying gate-level multipliers [31, 62, 80]. The practical algebraic calculus (PAC) [90] is a proof format to represent certificates for validating results of such algebraic techniques. It is based on the polynomial calculus (PC) [34] and allows us to capture that a polynomial can be derived from a given set of polynomials using algebraic ideal theory. In contrast to PC, PAC proofs can be checked efficiently, for example using our tool PACTRIM [90].

In this paper we add an indexing scheme to PAC and also propose deletion and extension rules. Our paper contains no new theory, beside the more technical formalization of extensions. This allows us to merge and check proofs obtained from SAT and Computer Algebra [63], the current state-of-the-art, in a uniform (and now precise) manner. The purpose of this system description is to define the new version of PAC and present our new checkers PACHECK and PASTÈQUE. Furthermore, PASTÈQUE in contrast to PACHECK is verified in Isabelle/HOL, but PACHECK is faster and more memory efficient (also compared to PACTRIM).

F.2 Practical Algebraic Calculus

In this section we briefly introduce the algebraic notion following [35]. Let X be the set of variables $\{x_1, \dots, x_n\}$ and further let $G \subseteq \mathbb{Z}[X]$ and $f \in \mathbb{Z}[X]$. Algebraic proof systems reason about polynomial equations. The aim is to show that the equation $f = 0$ is implied by the equations $g = 0$ for every $g \in G$, i.e., every common root of the polynomials $g \in G$ is also a root of f . In algebraic terms, this question means to derive whether f belongs to the ideal generated by G . A nonempty subset $I \subseteq \mathbb{Z}[X]$ is called an *ideal* if $\forall u, v \in I : u + v \in I$ and $\forall w \in \mathbb{Z}[X], \forall u \in I : wu \in I$. If $G = \{g_1, \dots, g_m\} \subseteq \mathbb{Z}[X]$, then the ideal generated by G is defined as $\langle G \rangle = \{q_1g_1 + \dots + q_mg_m \mid q_1, \dots, q_m \in \mathbb{Z}[X]\}$.

For a given set $G \subseteq \mathbb{Z}[X]$, a *model* is a point $u = (u_1, \dots, u_n) \in \mathbb{Z}^n$ such that $\forall g \in G : g(u_1, \dots, u_n) = 0$. Here, by $g(u_1, \dots, u_n)$ we mean the element of \mathbb{Z} obtained by evaluating the polynomial g for $x_1 = u_1, \dots, x_n = u_n$.

PAC proofs [90] are sequences of proof rules. We introduce the semantics of PAC as a transition system. Let P denote a sequence of polynomials, which can be accessed via indices. We write $P(i) = \perp$ to determine that the sequence P at index i does not contain a polynomial.

The initial state is $(X = \text{Var}(G \cup \{f\}), P)$ where P contains all polynomials of G . As already discussed [90] we are in general only interested in models of the Boolean domain, that is $x \in \{0, 1\}$ for $x \in X$. In our previous work, we added the set of Boolean-value constraints $B(X) = \{x^2 - x \mid x \in X\}$ to G and had to include steps in the proofs that operate on these Boolean-value constraints. Instead, we now handle operations on Boolean-value constraints implicitly to reduce the number of proof steps. That is, when checking the correctness, we immediately cancel exponents greater than

```

letter ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
number ::= '0' | '1' | ... | '9'
constant ::= (number)+
variable ::= letter (letter | number)*
term ::= variable ('*' variable)*
monomial ::= constant | [constant '*' ] term
polynomial ::= [ '-' ] monomial ('+' | '-' monomial)*
index ::= constant
input ::= (index polynomial ';' )*
add_rule ::= index '+' index ',' index ',' polynomial ';'
mul_rule ::= index '*' index ',' index ',' polynomial ';'
del_rule ::= index 'd' ';'
ext_rule ::= index '=' variable ',' polynomial ';'
proof ::= (add_rule | mul_rule | del_rule | ext_rule)*
target ::= polynomial ';'

```

Figure F.1: Syntax of input polynomials, target, and proofs in PAC-format.

one in the polynomials. The following two rules model the properties of ideals as introduced above.

ADD (i, j, k, p) $(X, P) \implies (X, P(i \mapsto p))$
 provided $P(j) \neq \perp$, $P(k) \neq \perp$, $P(i) = \perp$, and $p = P(j) + P(k) \in \mathbb{Z}[X]$.

MULT (i, j, q, p) $(X, P) \implies (X, P(i \mapsto p))$
 provided $P(j) \neq \perp$, $P(i) = \perp$, $q \in \mathbb{Z}[X]$, and $p = q \cdot P(j) \in \mathbb{Z}[X]$.

If in either one of the above rules p is also the target polynomial f , we know that $f \in \langle G \rangle$. In the original PAC format introduced in [90], it was necessary to explicitly provide the antecedents $P(i)$ and $P(j)$. In our new format, we use indices i and j to access polynomials, similar to LRAT [36]. The new syntax is given in Fig. F.1 and we provide an example in the appendix. Naming polynomials by indices reduces proof size and makes parsing more efficient, because only the conclusion polynomials of each rule and the initial polynomials of G have to be stated explicitly. However, introducing indices for polynomials has the effect that the semantics of P changes from sets to multisets, as in DRAT [42], and it becomes possible to introduce the same polynomial under different names.

We extend our original proof rules [90] by adding a deletion and an extension rule. In the deletion rule we remove polynomials from P which are not needed anymore in subsequent steps to reduce the memory usage of our tools.

DELETE(i) $(X, P) \implies (X, P(i \mapsto \perp))$

F.2.1 Extension

In our previous work [63], we converted DRUP proofs to the PAC format and encountered the need to extend the initial set of polynomials G to reduce the size of the polynomials in the PAC proof. We included polynomials of the form $-f_x + 1 - x$,

which introduced the variable f_x as the negation of the Boolean variable x . However, at that point we did not use proper extension rules, but simply added these extension polynomials to the initial polynomials G . This may affect the models, because any arbitrary polynomial can be added as initial constraints. To prevent this issue we add an extension rule to PAC, which preserves the original models on the original variable set X .

$$\text{EXT } (i, v, p) \quad (X, P) \implies (X \cup \{v\}, P(i \mapsto -v + p))$$

provided $P(i) = \perp$ and $v \notin X$ and $p \in \mathbb{Z}[X]$ and $p^2 - p = 0$.

With this extension rule, variables v can act as placeholders for polynomials p , i.e., $-v + p = 0$, which enables more concise proofs. The variables v are not allowed to occur earlier in the proof. Furthermore, to preserve Boolean models, we require $p^2 = p$ in order to guarantee that $v^2 = v$ holds. We provide the following fact (also proved in Isabelle but without using Gröbner bases theory).

Proposition F.1. *The rule EXT preserves the original models on X .*

Proof. We show that adding the polynomial $p_v := -v + p$ does not affect models of $P \subseteq \mathbb{Z}[X]$. For that we use the theory of Gröbner bases [25]. Let “ $<$ ” be a lexicographic term ordering, H a Gröbner basis of $\langle P \rangle$ w.r.t. “ $<$ ”, and “ $<_v$ ” be an extension of “ $<$ ” by adding v as largest element. Thm. 3 of [76] shows that $H \cup \{p_v\}$ is a Gröbner basis w.r.t. “ $<_v$ ” for $\langle P_v \rangle := \langle P(i \mapsto p_v) \rangle \subseteq \mathbb{Z}[X \cup \{v\}]$, the extended ideal, and $\langle P_v \rangle \cap \mathbb{Z}[X] = \langle H \cup \{p_v\} \rangle \cap \mathbb{Z}[X] = \langle H \rangle = \langle P \rangle$ follows. \square

F.3 Pacheck

We implemented PACHECK as an extension of PACTRIM [90]. It consists of approximately 1 700 lines of C code and is published [65] as open source under MIT license. The default mode supports the extended version of PAC, as presented in this paper, for the new syntax using indices. It also automatically reduces exponents. PACHECK is backwards compatible to our original format of PAC [90] and all features including reasoning with exponents are supported. However, extension rules are only supported for Boolean models.

PACHECK reads the three input files `<input>`, `<proof>`, and `<target>` and then verifies that the polynomial in `<target>` is contained in the ideal generated by the polynomials in `<input>` using the rules provided in `<proof>`. The polynomial arithmetic needed for checking the proof rules is implemented from scratch. In PACHECK polynomials are stored as ordered linked lists of monomials, where a monomial consists of a coefficient and a term. The coefficients are represented using the GMP library [44] for representing large integers. Terms are ordered linked list of variables (identified as strings).

We order variables in terms lexicographically using `strcmp`. All internally allocated terms in linked lists are shared using a hash table. It turns out that the order of variables has an enormous effect on memory usage, since different variable orderings induce

different terms (e.g., given the monomials xyz and $x'yz$, sharing of yz is possible for the order $x' > x > y > z$, whereas no sharing occurs for $y > x > z > x'$). For one example with more than 7 million proof steps, using `-lstrcmp` as sorting function leads to an increase of 50% in memory usage. Terms in polynomials are sorted lexicographically too.

In the initial phase of PACHECK each polynomial from `<input>` is sorted and stored as an inference. Inferences consist of a given index and a polynomial and are stored in a hash table. In the default mode, the index acts as the hash value. Thus it is possible to add the same polynomial twice. If the original format of PAC is used, a hash value is computed based on the input polynomial.

Proof checking is applied on-the-fly. We parse each rule of `<proof>` and immediately apply the necessary checks discussed in Sect. F.2. If the rule is either `ADD` or `MULT` we have to compute whether the conclusion polynomial of the rule is equal to the arithmetic operation performed on the antecedent polynomials.

We modified the algorithm of polynomial addition in PACTRIM and now assume the monomials of polynomials to be sorted. Addition of polynomials is performed by merging their monomials in an interleaved way. In PACTRIM we pushed the monomials of both polynomials on a stack and then sorted and merged them. Normalization of the exponents is not necessary in the `ADD` rule, but we still use this technique for multiplication of polynomials, where we multiply each monomial of the first polynomial with each monomial of the second monomial. In the `MULT` rule we normalize exponents larger than one, before testing equality. Furthermore, we check whether the conclusion polynomial of the rules `ADD` or `MULT` matches the polynomial in `<target>` in order to identify whether the target polynomial was derived.

The original version of PACTRIM [90] did not allow deletion of inferences. As a consequence the set of polynomials increased with each proof rule, leading to memory exhaustion for very large proofs. In PACHECK we now support deletion of inferences. A partial solution for deletion was already used [62] and lead to a drop of the memory usage. However, in contrast to our new version, individual inferences could not be deleted (only both antecedents of a proof step could be). Extension variables were not supported in PACTRIM [90] either.

F.4 Pastèque

To further increase trust in the verification, we implemented a verified checker called PASTÈQUE in the proof assistant Isabelle/HOL [84]. It follows a “refinement” approach, starting with an abstract specification of ideals, which we then refine with the Isabelle Refinement Framework [72] to the transition system from Sect. F.2, and further down to executable code using Isabelle’s code generator [45]. The Isabelle files have been made available [40]. The generated code consists of 2 800 lines Standard ML (2 400 generated by Isabelle, 400 for the parser) and is also available [65] under MIT license.

On the most abstract level, we start from Isabelle’s definition of ideals. The specification states that if “success” is returned, the target is in the ideal. Then we formalize

PAC and prove that the generated ideal is not changed by the rules. Proving that PAC respects the specification on ideals was not obvious due to limited automation and development of Isabelle library of polynomials (e.g., neither “ $\text{Var}(1) = \emptyset$ ” nor “ $p \neq 0 \implies X \in \text{Var}(X \times p)$ ” are present). However, Sledgehammer [18] automatically proved many of these simple lemmas.

While the input format identifies variables as strings, Isabelle only supports natural numbers as variables. Therefore, we use an injective function to convert between the abstract specification of polynomials (with natural numbers as variables) and the concrete manipulations (with strings as variables). The code does not depend on this function, only the correctness theorem does. Injectivity is only required to check that extension variables did not occur before.

In the third refinement stage, SEPREF [71] changes data structures automatically, such as replacing the set of variables X by a hash-set. Finally, we use the code generator to produce code. This code is combined with a trusted parser and can be compiled using the Standard ML compiler MLTON [98].

The implementation is less sophisticated than PACHECK’s. In particular, sharing is not considered (like sharing of variables in every polynomial) as it can be executed partially by the compiler, although this is not guaranteed by the Standard ML semantics. Some sharing could be performed by the garbage collector. We tried to enforce sharing by using MLTON’s `shareAll` function and by using a hash map during parsing¹, but performance was worse.

PASTÈQUE is one order of magnitude slower than PACHECK. First, this is due to Standard ML. While Isabelle’s code generator to LLVM [74] produces much faster code, we need integers of arbitrary large size, which is not supported currently, and sharing must be done entirely manually, which is challenging due to the separation logic used by SEPREF. Second, there is no axiomatization of file reading and hence parsing must be applied *entirely* before calling the checker in order for the correctness theorem to apply. This is more memory intensive and less efficient than interleaving parsing and checking. PASTÈQUE can be configured via the `uloop` option to either use the main loop generated by Isabelle (parsing before calling the generated checker) or instead use a hand-written copy of the main loop, the *unsafe loop*, where parsing and checking is interleaved and the checking functions are verified in Isabelle. The performance gain is large (on `sparc1-64` with 32 GB RAM, the garbage collection time decreased from over 700 s to 25 s), but the correctness theorem does not apply anymore.

F.5 Evaluation

In our experiments we used 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 rounded seconds (wall-clock time). We measure the wall-clock time from starting the tools until they are finished. In our experiments we aim to highlight the benefits of the new proof format and

¹We used a hash map that assigns a variable to “itself” (i.e., the same string, but potentially at a different memory location) and normalize every occurrence

multiplier	steps (10 ⁶)	deg	PACHECK						PASTÈQUE			
			no delete		no index		default		uloop			
			sec	MB	sec	MB	sec	MB	sec	MB	sec	MB
btor-128	0.4	3	5	273	11	100	5	92	22	3 886	17	1 773
btor-256	1.6	3	25	1 144	62	435	25	364	105	21 157	79	4 364
btor-512	6.3	3	138	4 956	402	1 972	141	1 461	531	64 412	416	22 292
sparrc-128	0.6	4	6	454	16	148	6	136	31	5 002	23	1 608
sparrc-256	2.3	4	29	1 858	96	651	27	541	139	32 525	102	8 769
sparrc-512	9.4	4	146	7 683	617	2 965	134	2 171	608	64 412	471	25 632
sparcl-32	1.6	256	23	773	35	353	21	352	122	39 107	116	7 667
spdtlf-32	0.3	46	2	122	3	73	2	72	11	1 657	11	1 054
bpctbk-32	0.2	25	1	86	2	51	1	51	8	1 546	7	1 030
bpwtcl-32	5.6	764	193	4 324	289	1 428	180	1 426	732	58 494	753	64 413

Table F.1: Proof Checking (in bold the fastest version).

provide a comprehensive comparison between our two tools. Source code, benchmarks and experimental data are available [65].

For the experiments of Table F.1 we generated PAC proofs as in previous work [62,63] in order to validate the correctness of multiplier circuits. The multipliers are either generated with the AMG [53], BOOLECTOR [83] or GENMUL [81].

For the upper part of Table F.1 we generated proof certificates with AMULET [62] to validate the correctness of simple multiplier circuits [62]. We modified AMULET to generate proofs in our new PAC format.

Our previous approach [62] to tackle complex multipliers relies on SAT solving (see also Sect. F.2.1) and requires to translate DRUP proofs into PAC [63] to obtain a single proof certificate. Experiments for these proof certificates are shown in the lower part of Table F.1. As already discussed, the proofs need extension rules and we modified the tools [63] to generate extension rules as presented in this paper. The second column shows the number of generated proof steps and the third the highest degree of the polynomials in the proof steps.

The effect of deleting rules and using indices in PACHECK can also be seen in Table F.1. Deletion rules reduce the memory usage by at least a factor two, although the effect on runtime is limited. Using indices reduces the runtime by around 30%. Note that in our earlier experiments [63] the proof checking time is slightly faster than in the column “no index”, because we did not use proper extension rules, which requires the additional checks $p \in \mathbb{Z}[X]$ and $p^2 = p$.

Furthermore, we can compare the performance of PACHECK and PASTÈQUE. The memory usage for PASTÈQUE depends on the garbage collector, which likely explains the peak around 64 GB (half of the available memory). The performance of the verified checker is sobering. PASTÈQUE is both much slower and more memory hungry.

Checkers of SAT certificates [48, 73] have the same level of efficiency as state-of-the-art checkers [88], likely because little to no garbage collection is required.

F.6 Conclusion and Future Work

We presented our proof checkers **PACHECK** and **PASTÈQUE** which are able to check PAC proofs efficiently. Our new proof format includes an extension rule, which is able to capture rewriting techniques. Furthermore, we added a deletion rule and used indices for polynomials. Our experiments showed that these optimizations cut memory usage in half and reduce the runtime by around 30%. **PACHECK** was four times faster than **PASTÈQUE** and used an order of magnitude less memory, whereas **PASTÈQUE** was formally verified in Isabelle.

In the future we want to capture more general extension rules in PAC as the calculus from Section F.2 allows. We imagine that it can be extended in two ways. First, we could relax the condition $p^2 = p$. This condition is necessary to have $v^2 = v$, but could be lifted even if it means that v^n cannot be simplified to v anymore, requiring to manipulate exponents. Second, we currently restrict the extension to the form $v = p$ where p contains no new variables. The correctness theorem does not rely on that and we leave it as future work to determine whether lifting one of these restrictions can lead to shorter proofs.

In the newest version of our tools [62] no redundant proof steps are generated, hence no backward proof checking is necessary unlike SAT certificates, but which might still be interesting in other applications. Another idea for future work is to bridge the gap between C and Isabelle, by verifying the C code directly.

F.7 Appendix

We show an example of a PAC proof, which demonstrates the usage of our tools.

Example F.2. Let $\bar{x} \vee \bar{y}$ and $y \vee z$ be two clauses. From these clauses we are able to derive the clause $\bar{x} \vee z$ using resolution. We show how this derivation can be covered in PAC.

At first we translate the clauses into polynomials using De Morgan's laws and using the fact that a logical AND can be represented by multiplication. For example, from $\bar{x} \vee \bar{y} = \top \Leftrightarrow x \wedge y = \perp$ we derive the polynomial equation $xy = 0$.

We translate the given clauses, which builds our input `<res.input>` and the target `<res.target>`. For the PAC proof in `<res.proof>` we introduce an extension variable f_z , which models the negation of z , i.e. $-f_z + 1 - z = 0$. We use this extension to reduce the size of the conclusion polynomials. The PAC proof shows only some possible deletion rules, adding more deletion rules is possible. The files of this example are available [65].

<pre><res.input> 1 x*y; 2 y*z-y-z+1; <res.target> -x*z+x;</pre>	<pre><res.proof> 3 = fz, -z+1; 4 * 3, y-1, -fz*y+fz-y*z+y+z-1; 5 + 2, 4, -fz*y+fz; 2 d; 4 d; 6 * 1, fz, fz*x*y; 1 d; 7 * 5, x, -fz*x*y+fz*x; 8 + 6, 7, fz*x; 9 * 3, x, -fz*x-x*z+x; 10 + 8, 9, -x*z+x;</pre>
--	--

We give these files to PACHECK and PASTÈQUE and these are the results:

```
$ pacheck res.input res.proof res.target
[pacheck] Pacheck Version 001
[pacheck] Practical Algebraic Calculus Proof Checker
[pacheck] Copyright (C) 2020, Daniela Kaufmann, JKU
[pacheck] compressed mode with indices assumed
[pacheck] checking target enabled
[pacheck] reading target polynomial from 'res.target'
[pacheck] read 8 bytes from 'res.target'
[pacheck] reading original polynomials from 'res.input'
[pacheck] found 2 original polynomials in 'res.input'
[pacheck] read 20 bytes from 'res.input'
[pacheck] reading polynomial algebraic calculus proof from
' res.proof'
[pacheck] found and checked 8 inferences in 'res.proof'
[pacheck] read 219 bytes from 'res.proof'
[pacheck] found 1 target polynomial inference
[pacheck] proof length 10 (number of polynomials)
```

```
[pacheck] proof size 25 (on average 2.5 terms per polynomial)
[pacheck] proof degree 3 (internal maximum degree 3)
[pacheck] searched 32 inferences 0.1 average collisions
[pacheck] 10 inferences, 3.2 average searches
[pacheck] original inferences 2 (20% of total rules)
[pacheck] inference rules 8 (80% of total rules)
[pacheck] addition inference rules 3 (38% of inference rules)
[pacheck] multiplication inference rules 4 (50% of inference
        rules)
[pacheck] extension rules 1 (12% of inference rules)
[pacheck] deletion inference rules 3 (30% of total rules)
[pacheck] maximum 9 of total 10 terms (90%)
[pacheck] searched 52 terms 81% hits 0.3 average collisions
[pacheck] maximum 2229 bytes allocated (0.0 MB)
[pacheck] maximum resident set size 4481024 bytes (4.3 MB)
[pacheck] process time 0.000 seconds
[pacheck] TARGET CHECKED
```

```
$ pasteque res.input res.proof res.target
c polys parsed
c *****
c pac parsed
c spec parsed
c Now checking
s SUCCESSFULL
c
c
c ***** stats *****
c parsing polys file init (nonGC):
  0.000 s = 0.000 s (usr) 0.000 s (sys)
c parsing pac file init (nonGC):
  0.000 s = 0.000 s (usr) 0.000 s (sys)
c full init (nonGC): 0.000 s = 0.000 s (usr) 0.000 s (sys)
c time solving (nonGC): 0.000 s = 0.000 s (usr) 0.000 s (sys)
c time GC: 0.000 s = 0.000 s (usr) 0.000 s (sys)
c time solving(full): 0.000 s
c Overall (nonGC): 0.001 s = 0.001 s (usr) 0.000 s (sys)
c overall GC: 0.000 s = 0.000 s (usr) 0.000 s (sys)
c Overall(full): 0.001 s
```

Bibliography

- [1] Erika Ábrahám, John Abbott, Bernd Becker, Anna Maria Bigatti, Martin Brain, Bruno Buchberger, Alessandro Cimatti, James H. Davenport, Matthew England, Pascal Fontaine, Stephen Forrest, Alberto Griggio, Daniel Kroening, Werner M. Seiler, and Thomas Sturm. Satisfiability checking and symbolic computation. *ACM Comm. Computer Algebra*, 50(4):145–147, 2016.
- [2] Snorri Agnarsson, Abdelilah Kandri-Rody, Deepak Kapur, Paliath Narendran, and B. David Saunders. Complexity of testing whether a polynomial ideal is nontrivial. In *MACSYMA User’s Conference*, pages 452–458, 1984.
- [3] Michael Alekhnovich, Eli Ben-Sasson, Alexander A. Razborov, and Avi Wigderson. Space Complexity in Propositional Calculus. *SIAM J. Comput.*, 31(4):1184–1211, 2002.
- [4] Michael Artin. *Algebra*. Prentice Hall, 1991.
- [5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- [6] Paul Beame, Russell Impagliazzo, Jan Krajíček, Toniann Pitassi, and Pavel Pudlák. Lower Bounds on Hilbert’s Nullstellensatz and Propositional Proofs. In *Proc. London Math. Society*, volume s3-73, pages 1–26, 1996.
- [7] Paul Beame and Vincent Liew. Towards Verifying Nonlinear Integer Arithmetic. In *International Conference on Computer Aided Verification, CAV 2017*, volume 10427 of *LNCS*, pages 238–258. Springer, 2017.
- [8] Paul Beame and Vincent Liew. Toward Verifying Nonlinear Integer Arithmetic. *J. ACM*, 66(3):22:1–22:30, 2019.
- [9] Thomas Becker, Volker Weispfenning, and Heinz Kredel. *Gröbner Bases*, volume 141 of *Graduate texts in mathematics*. Springer, 1993.
- [10] 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.
- [11] Armin Biere. PicoSAT Essentials. *JSAT*, 4:75–97, 2008.

Bibliography

- [12] Armin Biere. Collection of Combinational Arithmetic Miter Submitted to the SAT Competition 2016. In *SAT Competition 2016*, volume B-2016-1 of *Dep. of Computer Science - Series of Publications B*, pages 65–66. University of Helsinki, 2016.
- [13] Armin Biere. Splat, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016. In *SAT Competition 2016*, volume B-2016-1 of *Dep. of Computer Science - Series of Publications B*, pages 44–45. University of Helsinki, 2016.
- [14] Armin 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>.
- [15] Armin Biere. CaDiCaL at the SAT Race 2019. In *SAT Race 2019*, volume B-2019-1 of *Dep. of Computer Science - Series of Publications B*, pages 8–9. University of Helsinki, 2019.
- [16] Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 And Beyond. Technical report, FMV Reports Series, JKU Linz, Austria, 2011.
- [17] Armin Biere, Manuel Kauers, and Daniela Ritirc. Challenges in Verifying Arithmetic Circuits Using Computer Algebra. In *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017*, pages 9–15. IEEE Computer Society, 2017.
- [18] Jasmin C. Blanchette, Sascha Böhme, Mathias Fleury, Steffen Smolka, and Albert Steckermeier. Semi-intelligible Isar Proofs from Machine-Generated Proofs. *J. Autom. Reasoning*, 56(2):155–200, 2016.
- [19] Michael Brickenstein. *Boolean Gröbner bases – Theory, Algorithms and Applications*. PhD thesis, University of Kaiserslautern, 2011.
- [20] Michael Brickenstein and Alexander Dreyer. PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. *J. Symb. Comput.*, 44(9):1326 – 1345, 2009.
- [21] Curtis Bright, Ilias Kotsireas, and Vijay Ganesh. Applying Computer Algebra Systems and SAT Solvers to the Williamson Conjecture. *J. Symb. Comput.*, 2019. In press.
- [22] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.
- [23] Randal E. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. *IEEE Trans. Comput.*, 40(2):205–213, 1991.

- [24] Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits using binary moment diagrams. *STTT*, 3(2):137–155, 2001.
- [25] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, 1965.
- [26] Bruno Buchberger and Manuel Kauers. Gröbner basis. *Scholarpedia*, 5(10):7763, 2010. http://www.scholarpedia.org/article/Groebner_basis.
- [27] Josh Buresh-Oppenheim, Matthew Clegg, Russell Impagliazzo, and Toniann Pitassi. Homogenization and the polynomial calculus. *Computational Complexity*, 11(3-4):91–108, 2002.
- [28] Jiunn-Chern Chen and Yirng-An Chen. Equivalence Checking of Integer Multipliers. In *Asia and South Pacific Design Automation Conference, ASP-DAC 2001*, pages 169–174. ACM, 2001.
- [29] Yirng-An Chen and Randal E. Bryant. Verification of Arithmetic Circuits with Binary Moment Diagrams. In *Design Automation Conference, DAC 1995*, pages 535–541. ACM, 1995.
- [30] Yirng-An Chen, Edmund Clarke, Pei-Hsin Ho, Yatin Hoskote, Timothy Kam, Manpreet Khaira, John O’Leary, and Xudong Zhao. Verification of All Circuits in a Floating-Point Unit Using Word-Level Model Checking. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD 1996*, volume 1166 of *LNCS*, pages 19–33. Springer, 1996.
- [31] Maciej J. Ciesielski, Tiankai Su, Atif Yasin, and Cunxi Yu. Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model. *IEEE TCAD*, pages 1–1, 2019. Early acces.
- [32] Maciej J. Ciesielski, Cunxi Yu, Walter Brown, Duo Liu, and André Rossi. Verification of Gate-level Arithmetic Circuits by Function Extraction. In *Design Automation Conference, DAC 2015*, pages 52:1–52:6. ACM, 2015.
- [33] Edmund Clarke, Masahiro Fujita, and Xudong Zhao. Applications of Multi-Terminal Binary Decision Diagrams. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1995.
- [34] Matthew Clegg, Jeffery Edmonds, and Russell Impagliazzo. Using the Groebner Basis Algorithm to Find Proofs of Unsatisfiability. In *Symposium on Theory of Computing, STOC 1996*, pages 174–183. ACM, 1996.
- [35] David Cox, John Little, and Donal O’Shea. *Ideals, Varieties, and Algorithms*. Springer-Verlag New York, 1997.

- [36] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient Certified RAT Verification. In *International Conference on Automated Deduction, CADE-26*, volume 10395 of *LNCS*, pages 220–236. Springer, 2017.
- [37] Luís Cruz-Filipe, João Marques-Silva, and Peter Schneider-Kamp. Formally Verifying the Solution to the Boolean Pythagorean Triples Problem. *J. Autom. Reasoning*, 63(3):695–722, 2019.
- [38] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. SINGULAR 4-1-0 — A computer algebra system for polynomial computations. <http://www.singular.uni-kl.de>, 2016.
- [39] Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *International Conference on Theory and Applications of Satisfiability Testing, SAT 2005*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.
- [40] Mathias Fleury and Daniela Kaufmann. Isabelle Formalization of PAC. http://people.mpi-inf.mpg.de/~mfleury/IsaFoL/current/PAC_Checker/PAC_Checker/index.html, 2020. Theory files available at <https://bitbucket.org/isafol/isafol/src/master/PAC/>.
- [41] Allen Van Gelder. Verifying RUP Proofs of Propositional Unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008*, 2008.
- [42] Allen Van Gelder. Producing and verifying extremely large propositional refutations - Have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.
- [43] Evghenii I. Goldberg and Yakov Novikov. Verification of Proofs of Unsatisfiability for CNF Formulas. In *Design, Automation and Test in Europe Conference and Exposition, DATE 2003*, pages 10886–10891. IEEE Computer Society, 2003.
- [44] Torbjörn Granlund and the GMP development team. GNU MP: The GNU Multiple Precision Arithmetic Library, 2016. Version 6.1.2.
- [45] Florian Haftmann and Tobias Nipkow. Code Generation via Higher-Order Rewrite Systems. In *International Symposium on Functional and Logic Programming, FLOPS 2010*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.
- [46] Marijn J. H. Heule. Schur Number Five. *CoRR*, abs/1711.08076, 2017.
- [47] Marijn J. H. Heule and Armin Biere. Proofs for Satisfiability Problems. In *All about Proofs, Proofs for All Workshop, APPA 2014*, volume 55, pages 1–22. College Publications, 2015.

- [48] Marijn J. H. Heule, Warren A. Hunt, Jr., Matt Kaufmann, and Nathan Wetzler. Efficient, Verified Checking of Propositional Proofs. In *International Conference on Interactive Theorem Proving, ITP*, volume 10499 of *LNCS*, pages 269–284. Springer, 2017.
- [49] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while Checking Clausal Proofs. In *International Conference on Formal Methods in Computer Aided Design, FMCAD 2013*, pages 181–188. IEEE, 2013.
- [50] Marijn J. H. Heule, Manuel Kauers, and Martina Seidl. New ways to multiply 3×3 -matrices. *CoRR*, abs/1905.10192, 2019.
- [51] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer. In *International Conference on Theory and Applications of Satisfiability Testing, SAT 2016*, volume 9710 of *LNCS*, pages 228–245. Springer, 2016.
- [52] Edward Hirsch, Dmitry Itsykson, Arist Kojevnikov, Er Kulikov, and Sergey Nikolenko. Report on the Mixed Boolean-Algebraic Solver. Technical report, Laboratory of Mathematical Logic of St. Petersburg Dep. of Steklov Institute of Mathematics, 2005.
- [53] Naofumi Homma, Yuki Watanabe, Takafumi Aoki, and Tatsuo Higuchi. Formal Design of Arithmetic Circuits Based on Arithmetic Description Language. *IEICE Transactions*, 89-A(12):3500–3509, 2006.
- [54] Warren A. Hunt, Jr., Matt Kaufmann, J Strother Moore, and Anna Slobodova. Industrial hardware and software verification with ACL2. *Philos. Trans. Royal Soc. A*, 375(2104):20150399, 2017.
- [55] R. Impagliazzo, P. Pudlák, and J. Sgall. Lower bounds for the polynomial calculus and the Gröbner basis algorithm. *Computational Complexity*, 8(2):127–144, 1999.
- [56] Matti Järvisalo, Armin Biere, and Marijn J. H. Heule. Simulating Circuit-Level Simplifications on CNF. *J. Autom. Reasoning*, 49(4):583–619, 2012.
- [57] Abdelilah Kandri-Rody, Deepak Kapur, and Paliath Narendran. An Ideal-Theoretic Approach to Work Problems and Unification Problems over Finitely Presented Commutative Algebras. In *International Conference on Rewriting Techniques and Applications, RTA 1985*, volume 202 of *LNCS*, pages 345–364. Springer, 1985.
- [58] Deepak Kapur. Geometry theorem proving using Hilbert’s Nullstellensatz. In *Symposium on Symbolic and Algebraic Manipulation, SYMSAC 1986*, pages 202–208. ACM, 1986.

Bibliography

- [59] Deepak Kapur. Using Gröbner Bases to Reason About Geometry Problems. *J. Symb. Comput.*, 2(4):399–408, 1986.
- [60] Deepak Kapur and Paliath Narendran. An Equational Approach to Theorem Proving in First-Order Predicate Calculus. In *International Joint Conferences on Artificial Intelligence, IJCAI-85*, pages 1146–1153. Morgan Kaufmann, 1985.
- [61] Daniela Kaufmann, Armin Biere, and Manuel Kauers. Incremental Column-wise verification of arithmetic circuits using computer algebra. *Formal Methods in System Design*, 2019. Online first.
- [62] Daniela Kaufmann, Armin Biere, and Manuel Kauers. Verifying Large Multipliers by Combining SAT and Computer Algebra. In *International Conference on Formal Methods in Computer Aided Design, FMCAD 2019*, pages 28–36. IEEE, 2019.
- [63] Daniela Kaufmann, Armin Biere, and Manuel Kauers. From DRUP to PAC and back. In *Design, Automation and Test in Europe Conference and Exposition, DATE 2020*. IEEE, 2020. To appear.
- [64] Daniela Kaufmann, Armin Biere, and Manuel Kauers. SAT, Computer Algebra, Multipliers. In *Vampire 2018 and Vampire 2019. The 5th and 6th Vampire Workshops*, volume 71 of *EPiC Series in Computing*, pages 1–18. EasyChair, 2020.
- [65] Daniela Kaufmann and Mathias Fleury. The PAC checkers Pacheck and Pastèque. http://fmv.jku.at/pacheck_pasteque, 2020.
- [66] Daniela Kaufmann, Mathias Fleury, and Armin Biere. Pacheck and Pastèque, Checking Practical Algebraic Calculus Proofs. 2020. Submitted.
- [67] Daniela Kaufmann, Manuel Kauers, Armin Biere, and David Cok. Arithmetic Verification Problems Submitted to the SAT Race 2019. In *SAT Race 2019*, volume B-2019-1 of *Dep. of Computer Science - Series of Publications B*, page 49. University of Helsinki, 2019.
- [68] Matt Kaufmann and J. Strother Moore. ACL2 Version 8.2. <http://www.cs.utexas.edu/users/moore/acl2/>, 2019.
- [69] Israel Koren. *Computer Arithmetic Algorithms*. CRC Press, 2nd edition, 2001.
- [70] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE TCAD*, 21(12):1377–1394, 2002.
- [71] Peter Lammich. Refinement to Imperative/HOL. In *International Conference on Interactive Theorem Proving, ITP 2015*, volume 9236 of *LNCS*, pages 253–269. Springer, 2015.

- [72] Peter Lammich. Refinement Based Verification of Imperative Data Structures. In *International Conference on Certified Programs and Proofs, CPP 2016*, pages 27–36. ACM, 2016.
- [73] Peter Lammich. The GRAT Tool Chain - Efficient (UN)SAT Certificate Checking with Formal Correctness Guarantees. In *International Conference on Theory and Applications of Satisfiability Testing, SAT 2017*, volume 10491 of *LNCS*, pages 457–463. Springer, 2017.
- [74] Peter Lammich. Generating Verified LLVM from Isabelle/HOL. In *International Conference on Interactive Theorem Proving, ITP 2019*, volume 141 of *LIPICs*, pages 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
- [75] Massimo Lauria and Jakob Nordström. Graph Colouring is Hard for Algorithms Based on Hilbert’s Nullstellensatz and Gröbner Bases. In *Conference on Computational Complexity, CCC 2017*, volume 79 of *LIPICs*, pages 2:1–2:20. Schloss Dagstuhl, 2017.
- [76] Daniel Lichtblau. Effective computation of strong Gröbner bases over Euclidean domains. *Illinois Journal of Mathematics*, 56(1):177–194, 2012.
- [77] Jinpeng Lv and Priyank 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.
- [78] Jinpeng Lv, Priyank Kalla, and Florian Enescu. Efficient Gröbner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits. *IEEE TCAD*, 32(9):1409–1420, 2013.
- [79] Alireza Mahzoon, Daniel Große, and Rolf Drechsler. PolyCleaner: Clean your Polynomials before Backward Rewriting to verify Million-gate Multipliers. In *International Conference on Computer-Aided Design, ICCAD 2018*, pages 129:1 – 129:8. ACM, 2018.
- [80] Alireza Mahzoon, Daniel Große, and Rolf Drechsler. RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. In *Design Automation Conference, DAC 2019*, pages 185:1–185:6. ACM, 2019.
- [81] Alireza Mahzoon, Daniel Große, and Rolf Drechsler. Multiplier Generator GenMul. <http://www.sca-verification.org/>, 2019.
- [82] Mladen Miksa and Jakob Nordström. A Generalized Method for Proving Polynomial Calculus Degree Lower Bounds. In *Conference on Computational Complexity, CCC 2015*, volume 33 of *LIPICs*, pages 467–487. Schloss Dagstuhl, 2015.

Bibliography

- [83] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , BtorMC and Boolector 3.0. In *International Conference on Computer Aided Verification, CAV 2018*, volume 10981 of *LNCS*, pages 587–595. Springer, 2018.
- [84] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [85] Behrooz Parhami. *Computer Arithmetic - Algorithms and Hardware designs*. Oxford University Press, 2000.
- [86] Evgeny Pavlenko, Markus Wedler, Dominik Stoffel, Wolfgang Kunz, Oliver Wienand, and Evgeny Karibaev. Modeling of Custom-Designed Arithmetic Components for ABL Normalization. In *Forum on Specification and Design Languages, FDL 2008*, pages 124–129. IEEE, 2008.
- [87] Tim Pruss, Priyank Kalla, and Florian Enescu. Equivalence Verification of Large Galois Field Arithmetic Circuits using Word-Level Abstraction via Gröbner Bases. In *Design Automation Conference, DAC 2014*, pages 152:1–152:6. ACM, 2014.
- [88] Adrián Rebola-Pardo and Johannes Altmanninger. Frying the Egg, Roasting The Chicken: Unit Deletions in DRAT Proofs. In *International Conference on Certified Programs and Proofs, CPP 2020*, pages 61–70. ACM, 2020.
- [89] Daniela Ritirc, Armin Biere, and Manuel Kauers. Column-Wise Verification of Multipliers Using Computer Algebra. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD 2017*, pages 23–30. IEEE, 2017.
- [90] Daniela Ritirc, Armin Biere, and Manuel Kauers. A Practical Polynomial Calculus for Arithmetic Circuit Verification. In *Workshop on Satisfiability Checking and Symbolic Computation, SC2 2018*, pages 61–76. CEUR-WS, 2018.
- [91] Daniela Ritirc, Armin Biere, and Manuel Kauers. Improving and Extending the Algebraic Approach for Verifying Gate-Level Multipliers. In *Design, Automation and Test in Europe Conference and Exposition, DATE 2018*, pages 1556–1561. IEEE, 2018.
- [92] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.
- [93] Amr Sayed-Ahmed, Daniel Große, Ulrich Kühne, Mathias Soeken, and Rolf Drechsler. Formal verification of integer multipliers by combining gröbner basis with logic reduction. In *Design, Automation and Test in Europe Conference and Exposition, DATE 2016*, pages 1048–1053. IEEE, 2016.
- [94] Amr Sayed-Ahmed, Daniel Große, Mathias Soeken, and Rolf Drechsler. Equivalence checking using Gröbner bases. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD 2016*, pages 169–176. IEEE, 2016.

- [95] Dominik Stoffel and Wolfgang Kunz. Equivalence checking of arithmetic circuits on the arithmetic bit level. *IEEE TCAD*, 23(5):586–597, 2004.
- [96] Ashish Tiwari. An Algebraic Approach for the Unsatisfiability of Nonlinear Constraints. In *Computer Science Logic Workshop, CSL 2005*, pages 248–262. Springer, 2005.
- [97] Shobha Vasudevan, Vinod Viswanath, Robert W. Sumners, and Jacob A. Abraham. Automatic Verification of Arithmetic Circuits in RTL Using Stepwise Refinement of Term Rewriting Systems. *IEEE Trans. Comput.*, 56(10):1401–1414, 2007.
- [98] Stephen Weeks. Whole-Program Compilation in MLton. In *ACM Workshop on ML, 2006*, page 1. ACM, 2006.
- [99] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *International Conference on Theory and Applications of Satisfiability Testing, SAT 2014*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014.
- [100] Oliver Wienand, Markus Wedler, Dominik Stoffel, Wolfgang Kunz, and Gert-Martin 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.
- [101] Philipp Woelfel. Bounds on the OBDD-size of integer multiplication via universal hashing. *J. Computer and System Sciences*, 71(4):520 – 534, 2005.
- [102] Wolfram Research, Inc. Mathematica, 2016. Version 10.4.
- [103] Cunxi Yu, Walter Brown, Duo Liu, André Rossi, and Maciej J. Ciesielski. Formal Verification of Arithmetic Circuits by Function Extraction. *IEEE TCAD*, 35(12):2131–2142, 2016.
- [104] Cunxi Yu and Maciej 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.
- [105] Cunxi Yu and Maciej J. Ciesielski. Formal Analysis of Galois Field Arithmetic Circuits-Parallel Verification and Reverse Engineering. *IEEE TCAD*, 38(2):354–365, 2019.
- [106] Cunxi Yu, Maciej J. Ciesielski, and Alan Mishchenko. Fast Algebraic Rewriting Based on And-Inverter Graphs. *IEEE TCAD*, 37(9):1907–1911, 2018.
- [107] Lintao Zhang and Sharad Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *Design, Automation and Test in Europe Conference and Exposition, DATE 2003*, pages 10880–10885. IEEE, 2003.

Bibliography

- [108] Edward Zulkoski, Curtis Bright, Albert Heinle, Ilias S. Kotsireas, Krzysztof Czarnecki, and Vijay Ganesh. Combining SAT Solvers with Computer Algebra Systems to Verify Combinatorial Conjectures. *J. Autom. Reasoning*, 58(3):313–339, 2017.

Curriculum Vitae

Personal Details

Name: Daniela Kaufmann
formerly Daniela Ritirc

Born: 1991, Linz, Austria

Research

2016 – 2020 Research and Teaching Assistant
Johannes Kepler University, Linz, Austria
Institute for Formal Models and Verification (FMV)

Work

2014 – 2016 Student Assistant
Johannes Kepler University, Linz, Austria

Education

2016 – 2020 PhD Student in Computer Science,
Johannes Kepler University, Linz, Austria
Advisor: Univ.-Prof. Dr. Armin Biere

2014 – 2016 Master Studies in Computer Mathematics,
Johannes Kepler University, Linz, Austria

2011 – 2014 Bachelor Studies in Technical Mathematics,
Johannes Kepler University, Linz, Austria

2006 – 2011 Higher Institute for Tourism and Catering
Höhere Lehranstalt für Tourismus,
Bad Leonfelden, Austria

2002 – 2006 Grammar School
Bundesrealgymnasium, Linz, Austria

1998 – 2002 Primary School
Volksschule, Linz, Austria