# Polynomial Formal Verification of Arithmetic Circuits

Rolf Drechsler[1,2], Alireza Mahzoon[1], and Lennart Weingarten[2]

[1] Institute of Computer Science, University of Bremen, Bremen, Germany
[2] Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{drechsle, mahzoon, len_wei}@uni-bremen.de

**Abstract.** The size and the complexity of digital circuits are increasing rapidly. This makes the circuits highly error-prone. As a result, proving the correctness of a circuit is of utmost importance after the design phase. Arithmetic circuits are among the most challenging designs to verify due to their high complexity and big size. In recent years, several formal methods have been proposed to verify arithmetic circuits. However, the time and space complexity bounds are still unknown for most of these approaches, resulting in performance unpredictability. In this paper, we clarify the importance of polynomial formal verification for digital designs particularly arithmetic circuits. We also introduce an *Arithmetic Logic Unit* (ALU) and prove that formal verification of this circuit is possible in polynomial time. Finally, we confirm the correctness of the complexity bounds by experimental results.

**Keywords:** Formal Verification, Complexity, Polynomial, Arithmetic Circuits, Arithmetic Logic Unit, Binary Decision Diagram

## 1 Introduction

With the invention of the transistor back in 1947, the cornerstone for the digital revolution was laid. As a fundamental building block, the transistor enabled the development of digital circuits. Their mass production revolutionized the field of electronics, finally leading to computers, embedded systems, as well as the internet. Hence, the impact of digital hardware on society, as well as economy, was and is tremendous. Over the last decades, the enormous growth of complexity of integrated circuits continues as expected. As modern electronic devices are getting more and more ubiquitous, the fundamental issue of functional correctness becomes more important than ever. This is evidenced by many publicly known examples of electronic failures with disastrous consequences. This includes e.g. the Intel Pentium bug in 1994, the New York blackout in 2003, and a design flaw in Intels Sandy Bridge chipset in 2011.

Such costly mistakes can only be prevented by applying rigorous verification to the circuits before they get to production [4,5]. A lot of effort has been put into developing efficient verification techniques by both academic and industrial research. Only recently, the industry has recognized the great importance of automated formal verification (see e.g. functional safety standards such as ISO 26262 [30]). Hence, in the last few years, this research area has become increasingly active. Essentially, the aim of automated formal verification is to automatically prove that an implementation is correct with respect to its specification. Depending on what is an implementation and what comprises a specification, different verification problems arise.

Automated formal verification of arithmetic circuits is one of the most important goals of the verification community in recent years. Arithmetic circuits are usually involved in many applications that require intense computations (e.g., cryptography and signal processing) as well as in architectures for artificial intelligence (e.g., machine learning). They include a wide range of different designs like adders, subtractors, multipliers, and dividers. The function of an arithmetic unit is unique, e.g., a multiplier always computes the product of its inputs. However, the architecture and size of them vary based on the application and design parameters. Most of these circuits are highly parallel and architecturally complex [17,32]. Therefore, the high complexity and the big size of arithmetic circuits make them very challenging to verify.

During the last 20 years, researchers have proposed several formal methods to verify arithmetic circuits: (a) *Binary Decision Diagram* (BDD) [2,10] verification methods extract the BDDs for the outputs of an arithmetic circuit with the help of symbolic simulation, then, the output BDDs are evaluated for the correctness, (b) *Boolean Satisfiability* (SAT) [11,3] verification methods translate the implementation and the specification into one Conjunctive Normal Form (CNF) which is satisfiable if the implementation is correct, (c) *Binary Moment Diagram* (*BMD and K*BMD) [8,12] approaches use word-level graphs to prove the equivalence between the word-level design specification and the bit-level circuit, (d) term rewriting [13,28,27] techniques take advantage of a library of rewrite rules to prove the correctness of arithmetic circuit by several rewritings in a theorem proving system, (e) reverse engineering techniques using *Arithmetic Bit-Level* (ABL) [26,24] extract an arithmetic bit-level description of the circuit, and then use it for a fast equivalence checking, and (f) *Symbolic Computer Algebra* (SCA) [20,21,23,15,22] methods capture the logical gates as polynomials, then, prove the correctness of arithmetic circuit by dividing the word-level specification by the gate polynomials.

Despite the success of formal verification approaches in proving the correctness of a wide variety of arithmetic circuits, the complexity bounds for most of these approaches are still unknown. It raises serious questions about the scalability of the verification methods. Furthermore, unknown time and space complexities make it difficult to compare two approaches and choose the best one with respect to the type of arithmetic circuit, e.g., which formal method is suitable for verification of an integer adder. In this paper, we first clarify the importance of polynomial formal verification. Then, we review the known researches about polynomial formal verification of arithmetic circuits. Subsequently, we introduce a simple *Arithmetic Logic Unit* (ALU) with 8 operations and prove that its formal verification is possible in polynomial time. For the first time, we also calculate the complexity bounds for verifying a subtractor (i.e., one of the ALU units). We confirm the correctness of obtained complexity bounds by experimental results. Finally, we propose a verification strategy for more advanced ALUs performing complicated operations such as multiplication.

## 2 Polynomial Formal Verification

In this section, we first clarify the importance of polynomial formal verification. Subsequently, we review the three important works in the field of polynomial formal verification of arithmetic circuits.

### 2.1 Importance of Polynomial Verification

The state-of-the-art formal verification techniques often give satisfying results for a specific type of arithmetic circuits: (a) BDD and SAT-based verification methods report very good results for different types of adder architectures, (b) *BMDs are used to verify multipliers, and (c) SCA-based approaches are employed for the verification of complex multipliers and dividers.

However, the main shortcoming of these techniques is the unpredictability in performance, leading to several verification problems:

1. It cannot be predicted before actually invoking the verification tool whether it will successfully terminate or run for an indefinite amount of time; e.g., it is not clear whether SAT-based verification can successfully verify all types of adders or it runs forever for some of them.
2. The scalability of these techniques remains unknown, i.e., it is not predictable how much the run-time and the required memory increase when the size of the circuit under verification grows; e.g., it is not obvious how scalable is a *BMD-based verification technique when it comes to proving the correctness of an integer multiplier.
3. It is not possible to compare the performance of verification methods for a specific design and choose the best one, e.g., it is not clear which verification technique has a better performance when it comes to proving the correctness of an integer adder.

In order to resolve the unpredictability of a verification method, its time and space complexity should be calculated. Knowing the complexity bounds for a verification technique alleviates the three aforementioned verification problems. We are particularly interested in optimized time and space bounds with the smallest possible polynomial order, i.e., $\mathcal{O}(n^m)$ where $n$ is the number of input bits and $m$ is a positive number. A formal verification method with a polynomial complexity (time and space) is scalable and can be carried out successfully.

In the next section, we review the three works in the field of polynomial formal verification of arithmetic circuits and illustrate their advantages and disadvantages.

### 2.2 Research Works in Polynomial Verification

During the last 20 years, researchers have proposed several formal verification methods to verify different types of arithmetic circuits. Despite rapid progress in developing formal methods, the research on the complexity bounds of verification methods is very limited. We now review the three notable works on the polynomial formal verification of arithmetic circuits.

**Polynomial Formal Verification of Adders using BDDs:** It is known for a long time that BDDs are very efficient in practice when it comes to the verification of adders. However, it has not been proven theoretically until recently. PolyAdd [6] showed for the first time that the complete formal verification process for three types of adders (i.e., ripple carry adder, conditional sum adder, and carry look-ahead adder) can be carried out polynomially. It was achieved by proving that the size of BDDs remains polynomial with respect to the number of input bits during the symbolic simulation. Although PolyAdd successfully shows that the polynomial formal verification of the three adder architectures is possible, it has two limitations: The proofs cannot be extended to the other adders, e.g. parallel prefix adders, and it does not calculate the exact order of complexity bounds. Recently, some progress has been made in calculating the exact verification complexities of adder architectures, e.g., the exact complexity bounds have been obtained for ripple carry adder and conditional sum adder in [18,19].

**Polynomial Formal Verification of Multipliers using *BMDs:** Verification of multipliers became possible after the development of word-level decision diagrams. Particularly, *BMDs and K*BMDs reported very good results for verifying several types of integer multipliers. However, [16] is the only work on the polynomial formal verification of multipliers using *BMDs. In the paper, the authors analyzed *BMD-based verification by backward construction applied to the class of Wallace-tree-like multipliers. They formally proved polynomial upper bounds on run-time and space requirements with respect to the number of input bits. They showed that the whole verification process is bounded by $\mathcal{O}(n^2)$ and $\mathcal{O}(n^4)$ with respect to space and time, where $n$ in the number of input bits. The proof in the paper is only for Wallace-tree-like multipliers and it does not support other classes of multipliers.

**Polynomial Formal Verification of Multipliers using SCA:** Recently, the SCA-based verification methods have been very successful in proving correctness of a large range of multiplier architectures. AMulet [15] is one of the SCA-based methods that is developed for verification of complex multipliers. The authors have shown in [14] that AMulet has a polynomial complexity when it comes to the verification of btor-multipliers (Generated by Boolector) and Wallace-tree multipliers with Booth encoding. However, the proof for the second multiplier

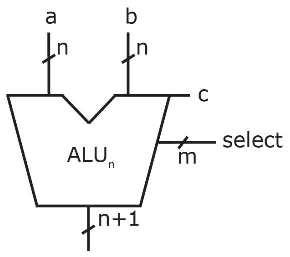

Fig. 1: Symbolic representation of the ALU

type is empirical, and it has not been done theoretically due to the irregular structure of the multiplier.

In addition to these three works, some research works have been recently done on polynomial BDD construction of totally symmetric functions [9] and polynomial formal verification of tree-like circuits [7].

## 3 Polynomial Formal Verification of a simple ALU

In this section, we first introduce a simple ALU with 8 operations. Then, we give a brief overview of the BDD-based verification. Subsequently, we calculate the time complexity of verifying the ALU and show that polynomial formal verification is possible for this architecture. Finally, we confirm our theoretical calculations by experimental results.

### 3.1 ALU Overview

An ALU is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers. The type and the number of supported operations in an ALU depend on the application. Fig. 1 shows the symbolic representation of a general ALU. It receives two $n$-bit inputs $a$ and $b$ as well as an input carry $c$. The operation between the inputs is determined by an $m$-bit *select*.

In this paper, we consider a simple ALU with 8 operations, i.e. the *select* signal has 3 bits. The complete list of supported operations is depicted in Table 1. The ALU can perform two arithmetic operations (i.e., addition and subtractions) as well as three bitwise logical operations (i.e., XOR, OR, and AND). The detailed architecture of the ALU is shown in Fig. 2.

### 3.2 BDD-based Verification

**Definition 1.** *A* Binary Decision Diagram *(BDD) is a directed, acyclic graph. Each node of the graph has two edges associated with the values of the variables 0 and 1. A BDD contains two terminal nodes (leaves) that are associated with the values of the function 0 or 1.*

Table 1: List of supported operations

| $s_2$ | $s_1$ | $s_0$ | function |
|---|---|---|---|
| 0 | 0 | 0 | $0 \ldots 0$ |
| 0 | 0 | 1 | $b - a$ |
| 0 | 1 | 0 | $a - b$ |
| 0 | 1 | 1 | $a + b + c$ |
| 1 | 0 | 0 | $a \oplus b$ |
| 1 | 0 | 1 | $a \vee b$ |
| 1 | 1 | 0 | $a \wedge b$ |
| 1 | 1 | 1 | $1 \ldots 1$ |

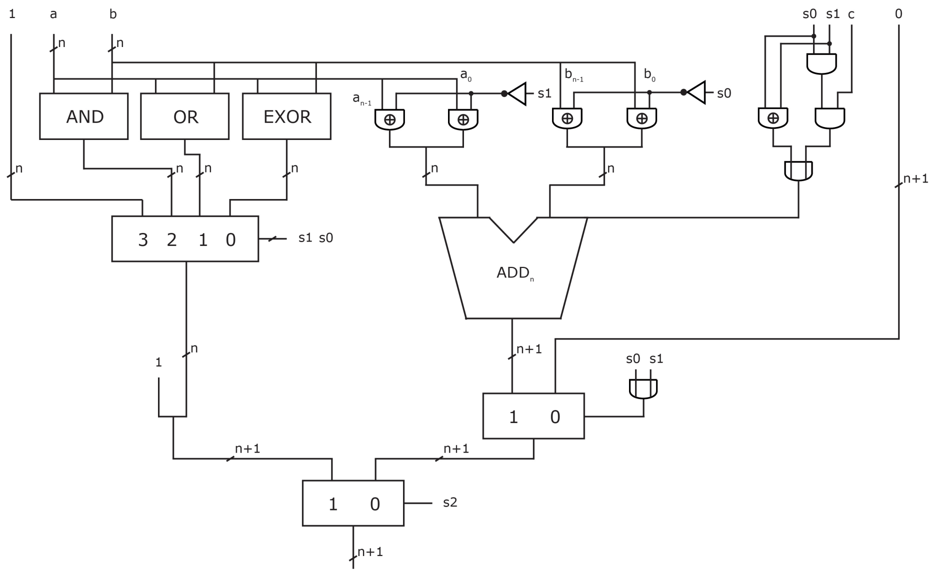

Fig. 2: ALU architecture

**Definition 2.** *An* Ordered Binary Decision Diagram *(OBDD) is a BDD, where different variables appear in the same order in each path from the root to a leaf.*

**Definition 3.** *A* Reduced Ordered Binary Decision Diagram *(ROBDD) is an OBDD that has a minimum number of nodes for a given variable order. The ROBDD of a Boolean function is always unique.*

The ITE operator (If-Then-Else) is used to calculate the results of the logical operations in BDDs:

$$ITE(f, g, h) = (f \wedge g) \vee (\bar{f} \wedge h) \tag{1}$$

**Algorithm 1** If-Then-Else (ITE)

```
Input: $f$, $g$, $h$ BDDs
Output: ITE BDD
 1: if terminal case then
 2:     return $result$
 3: else if computed-table has entry $\{f, g, h\}$ then
 4:     return $result$
 5: else
 6:     $v$ = top variable for $f$, $g$, or $h$
 7:     $t = ITE(f_{v=1}, g_{v=1}, h_{v=1})$
 8:     $e = ITE(f_{v=0}, g_{v=0}, h_{v=0})$
 9:     $R = FindOrAddUniqueTable(v, t, e)$
10:     $InsertComputedTable(\{f, g, h\}, R)$
11:     return $R$
```

The basic binary operations can be translated into the ITE operator:

$$\begin{aligned} f \wedge g &= ITE(f, g, 0), \\ f \vee g &= ITE(f, 1, g), \\ f \oplus g &= ITE(f, \overline{g}, g), \\ f \odot g &= ITE(f, g, \overline{g}), \\ \bar{f} &= ITE(f, 0, 1) \end{aligned} \tag{2}$$

ITE can be also used recursively in order to compute the results:

$$ITE(f, g, h) = ITE(x_i, ITE(f_{x_i}, g_{x_i}, h_{x_i}), ITE(f_{\overline{x}_i}, g_{\overline{x}_i}, h_{\overline{x}_i})) \tag{3}$$

where $f_{x_i}$ ($f_{\overline{x}_i}$) is the positive (negative) cofactor of $f$ with respect to $x_i$, i.e., the result of replacing $x_i$ by the value 1 (0).

ITE operations can be computed with the help of Algorithm 1. The result is obtained recursively based on Eq. (3) in this algorithm. During the calculations, the sub-diagrams of $f$, $g$ and $h$ are the arguments for subsequent calls to the ITE subroutine. The number of sub-diagrams in a BDD is equal to the number of nodes. For each of the three arguments, the sub-routine is called at most once. Assuming that *Unique Table* is searched at a constant time, the computational complexity of the ITE algorithm, even in the worst-case, does not exceed $\mathcal{O}(|f| \cdot |g| \cdot |h|)$, where $|f|$, $|g|$ and $|h|$ denote the size of the BDDs in terms of the number of nodes [1].

In order to formally verify a circuit, we need to have the BDD representation of the outputs. Symbolic simulation helps us to obtain the BDD for each primary output. During a simulation, an input pattern is applied to a circuit, and the resulting output values are checked to see whether they match the expected values. On the other hand, symbolic simulation verifies a set of scalar tests (which usually cover the whole input space) with a single symbolic test. Symbolic simulation using BDDs is done by generating corresponding BDDs for the input signals. Then, starting from primary inputs, the BDD for the output of a gate

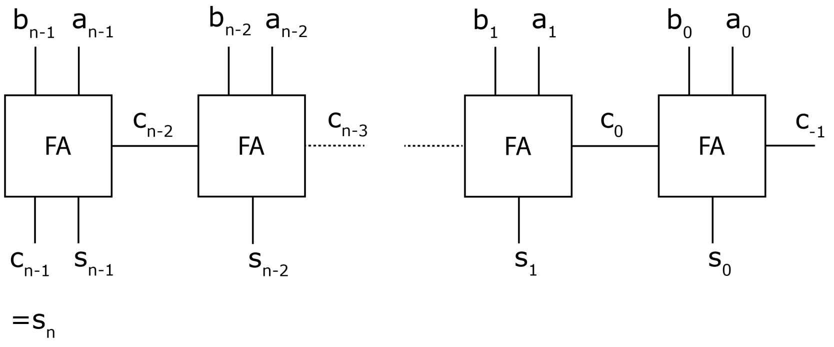

Fig. 3: Ripple carry adder

(or a building block) is obtained using the ITE algorithm. This process continues until we reach the primary outputs. Finally, the output BDDs are evaluated to see whether they match the BDDs of the circuit.

We now prove that verification of the introduced ALU is possible in polynomial time if we use the BDD-based verification. To do this, we should first calculate the complexity bounds for each ALU operation.

## 3.3 Complexity Bounds of Arithmetic Units

The introduced ALU can perform two arithmetic operations, i.e., addition and subtraction. We first focus on addition. We assume that the adder is a ripple carry adder as shown in Fig. 3.

In order to obtain the computational complexity of an $n$-bit ripple carry adder, we first calculate the complexity of BDD construction for a single FA. The sum and carry bits of a FA can be shown by ITE operations as follows:

$$\begin{aligned} S_i &= A_i \oplus B_i \oplus C_{i-1} = ITE(C_{i-1}, A_i \odot B_i, A_i \oplus B_i) = \\ &\quad ITE(C_{i-1}, ITE(A_i, B_i, \overline{B}_i), ITE(A_i, \overline{B}_i, B_i)), \qquad (4)\\ C_i &= (A_i \wedge B_i) \vee (A_i \wedge C_{i-1}) \vee (B_i \wedge C_{i-1}) = ITE(C_{i-1}, A_i \vee B_i, A_i \wedge B_i) = \\ &\quad ITE(C_{i-1}, ITE(A_i, 1, B_i), ITE(A_i, B_i, 0)) \qquad (5) \end{aligned}$$

The ITE operations are computed by Algorithm 1 to get the BDDs for the $S_i$ and $C_i$ signals. Assuming that $f$, $g$ and $h$ are the input arguments of an ITE operator, the computational complexity is computed as $|f| \cdot |g| \cdot |h|$. As a result, the complexity of computing $S_i$ and $C_i$ is as follows:

$$Complexity(S_i) = |C_{i-1}| \cdot |A_i|^2 \cdot |B_i|^2 \cdot |\overline{B}_i|^2 = 729 \cdot |C_{i-1}| \qquad (6)$$

$$Complexity(C_i) = |C_{i-1}| \cdot |A_i|^2 \cdot |B_i|^2 = 81 \cdot |C_{i-1}| \qquad (7)$$

where $A_i$, $B_i$, and $\overline{B}_i$ BDDs have only one internal node and two terminal nodes; thus, the size of them is the same and equals 3.

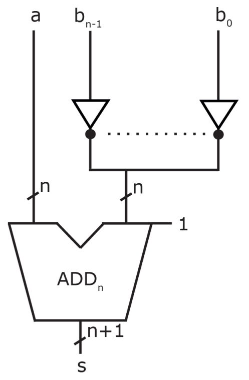

Fig. 4: Subtractor structure

It has been proven in [29] that the BDD size of the $i^{th}$ carry bit $(C_i)$ is $3 \cdot i + 6$. Thus, the overall complexity of verifying a ripple carry adder can be obtained as follows:

$$complexity_{[RCA]} = 810 \cdot \sum_{i=0}^{n-1} |C_{i-1}| = 2430 \cdot \sum_{i=0}^{n-1} (i+1) = 1215n^2 + 1215n \quad (8)$$

We can conclude that the order of the verification complexity is $\mathcal{O}(n^2)$, where $n$ is the number of bits per input (i.e., size of the adder). As a result, proving the correctness of a ripple carry adder has quadratic time complexity.

Fig. 4 shows the structure of a subtractor. In a subtractor architecture, the bits of one of the inputs are negated and the carry signal is set to one. The time complexity of negating $n$ bits is of linear order with respect to the number of input bits. After the negation, the size of the BDD does not change, thus, the same calculations for obtaining the complexity bounds of an adder are applicable to a subtractor. As a result, formal verification of a subtractor has quadratic time complexity, i.e., $\mathcal{O}(n^2)$.

## 3.4 Complexity Bounds of Logic Units

The introduced ALU has three bitwise logical operations, i.e., XOR, OR and AND. Each bitwise operation is done by $n$ gates. The time complexity of obtaining the output BDD of a gate (e.g., $a_i \oplus b_i$) is constant. Thus, the overall time complexity is linear (i.e., $\mathcal{O}(n)$) with respect to the number of input bits.

The arithmetic units have a bigger order of time complexity in comparison to the logic units. Therefore, they determine the overall bounds. Consequently, verification of the simple ALU has quadratic time complexity, i.e., $\mathcal{O}(n^2)$.

## 3.5 Experimental Results

We have implemented the ALU in Fig. 2 in Verilog. The size of the ALU is a parameter and it can be set before the synthesis. Thus, we can easily generate

Table 2: Run-time of verifying ALUs (seconds)

| **Size** | **Run-time** |
|---:|---:|
| 1024 | 37.51 |
| 2048 | 83.51 |
| 3072 | 100.67 |
| 4096 | 111.30 |
| 5120 | 129.93 |
| 6144 | 146.82 |
| 7168 | 160.25 |
| 8192 | 171.81 |
| 9216 | 187.57 |
| 10240 | 206.79 |

ALUs of different sizes. The design has been synthesized using Yosys [31]. We have also implemented the BDD-based verifier in C++. The tool takes advantage of the symbolic simulation to obtain the BDDs for the primary outputs. In order to handle the BDD operations, we used the CUDD library [25]. All experiments are performed on an Intel(R) Xeon(R) CPU E3-1275 with 3.60 GHz and 64 GByte of main memory.

In order to verify the ALU, we first set a value to *select* signal (e.g., 011), then, we obtain the output BDDs using our verifier. Finally, we evaluate the BDDs to check whether they match the corresponding operation (e.g., addition). We repeat the process for all possible values of *select* to cover all operations.

Table 2 reports the verification run-times for simple ALUs. We have done the experiments for 10 ALUs of different sizes. The first column **Size** denotes the size of the ALU based on the number of bits per input. The run-time (in seconds) of the BDD-based verification method is reported in the second column **Run-time**.

It is evident in Table 2 that the BDD-based based verification reports very good results. An ALU with 10240 bits per input, which consists of more than 700K gates, can be verified in less than 4 minutes. Thus, the experimental results for the simple ALU confirm the scalability of the BDD-based verification method.

## 4 Polynomial Formal Verification of Advanced ALUs

In the previous section, we proved that polynomial formal verification of a simple ALU is possible. However, the ALUs which are used in practice are more complicated, and they contain arithmetic units such as multipliers. Thus, the BDD-based verification fails for these advanced ALUs, as it has an exponential time and space complexity for multiplier units. It is a general problem of monolithic verification strategies in which only one formal verification technique is used to prove the correctness of the whole design.

As we mentioned in Section 2.1, one of the advantages of knowing the complexity bounds is the possibility of comparing verification methods and choosing the best one for a specific design. When it comes to the verification of an advanced ALU, we can use different verification methods to prove the correctness of each operation based on the *select* signal; i.e., a verification method with the smallest order of complexity is employed for verifying a specific operation. This hybrid verification strategy (i.e., using different verification methods to verify each operation) makes the polynomial formal verification of an advanced ALU possible. Moreover, it also keeps the run-time and memory usage low.

As an example, a hybrid verification strategy can be used to verify an advanced ALU containing a multiplier unit. *BMD-based verification method reported good results for proving the correctness of multipliers, and its time and space complexity are polynomial [16]. On the other hand, we proved that polynomial formal verification of adder, subtractor and logic units is possible using BDD-based verification. Consequently, we use *BMD-based verification for the multiplication and the BDD-based verification for the rest of the operations.

In the future, we plan to calculate the complexity bounds of more verification methods. It helps us to compare the complexities and choose the best technique for the verification of a specific operation. As a result, the hybrid verification achieves better performance in terms of run-time and memory usage.

## 5 Conclusion

In this paper, we clarified the problem of performance unpredictability in the field of formal verification. We then discussed the importance of polynomial formal verification of arithmetic circuits and reviewed the most notable works in this research area. Subsequently, as an example, we proved the complexity bounds of a simple ALU using BDD-based verification and confirmed the theoretical calculations by experimental results. Finally, we proposed the idea of hybrid verification to prove the correctness of advanced ALUs containing complicated arithmetic units such as multipliers.

**Acknowledgements** This work was supported by the German Research Foundation (DFG) within the Reinhart Koselleck Project *PolyVer: Polynomial Verification of Electronic Circuits* (DR 287/36-1).

## References

1. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: Design Automation Conf. pp. 40–45 (1990)
2. Bryant, R.E.: Binary decision diagrams and beyond: enabling technologies for formal verification. In: International Conference on Computer-Aided Design. pp. 236–243 (1995)
3. Disch, S., Scholl, C.: Combinational equivalence checking using incremental SAT solving, output ordering, and resets. In: ASP Design Automation Conf. pp. 938–943 (2007)
4. Drechsler, R.: Advanced Formal Verification. Kluwer Academic Publishers (2004)

5. Drechsler, R.: Formal System Verification: State-of the-Art and Future Trends. Springer (2017)
6. Drechsler, R.: PolyAdd: Polynomial formal verification of adder circuits. In: IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems. pp. 99–104 (2021)
7. Drechsler, R.: Polynomial circuit verification using BDDs (2021), arXiv:2104.03024
8. Drechsler, R., Becker, B., Ruppertz, S.: The K*BMD: A verification data structure. IEEE Design & Test of Computers **14**(2), 51–59 (1997)
9. Drechsler, R., Dominik, C.: Edge verification: Ensuring correctness under resource constraints. In: Symposium on Integrated Circuits and System Design (2021)
10. Drechsler, R., Sieling, D.: Binary decision diagrams in theory and practice. Int. J. Softw. Tools Technol. Transf. **3**(2), 112–136 (2001)
11. Goldberg, E.I., Prasad, M.R., Brayton, R.K.: Using SAT for combinational equivalence checking. In: Design, Automation and Test in Europe. pp. 114–121 (2001)
12. Höreth, S., Drechsler, R.: Formal verification of word-level specifications. In: Design, Automation and Test in Europe. pp. 52–58 (1999)
13. Kapur, D., Subramaniam, M.: Mechanical verification of adder circuits using rewrite rule laboratory. Formal Methods in System Design **13**(2), 127–158 (1998)
14. Kaufmann, D., Biere, A.: Nullstellensatz-proofs for multiplier verification. In: Computer Algebra in Scientific Computing. Lecture Notes in Computer Science, vol. 12291, pp. 368–389. Springer (2020)
15. Kaufmann, D., Biere, A., Kauers, M.: Verifying large multipliers by combining SAT and computer algebra. In: Int'l Conf. on Formal Methods in CAD. pp. 28–36 (2019)
16. Keim, M., Drechsler, R., Becker, B., Martin, M., Molitor, P.: Polynomial formal verification of multipliers. Formal Meth. in Sys. Des. **22**(1), 39–58 (2003)
17. Koren, I.: Computer Arithmetic Algorithms. A. K. Peters, Ltd., 2nd edn. (2001)
18. Mahzoon, A., Drechsler, R.: Late breaking results: Polynomial formal verification of fast adders. In: Design Automation Conf. (2021)
19. Mahzoon, A., Drechsler, R.: Polynomial formal verification of area-efficient and fast adders. In: Reed-Muller Workshop (2021)
20. Mahzoon, A., Große, D., Drechsler, R.: PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers. In: International Conference on Computer-Aided Design. pp. 129:1–129:8 (2018)
21. Mahzoon, A., Große, D., Drechsler, R.: RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers. In: Design Automation Conf. pp. 185:1–185:6 (2019)
22. Mahzoon, A., Große, D., Drechsler, R.: Revsca-2.0: SCA-based formal verification of non-trivial multipliers using reverse engineering and local vanishing removal. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2021)
23. Mahzoon, A., Große, D., Scholl, C., Drechsler, R.: Towards formal verification of optimized and industrial multipliers. In: Design, Automation and Test in Europe. pp. 544–549 (2020)
24. Pavlenko, E., Wedler, M., Stoffel, D., Kunz, W., Wienand, O., Karibaev, E.: Modeling of custom-designed arithmetic components in ABL normalization. In: Forum on Specification and Design Languages. pp. 124–129 (2008)
25. Somenzi, F.: CUDD: CU decision diagram package release 2.7.0. available at `https://github.com/ivmai/cudd` (2018)

26. Stoffel, D., Kunz, W.: Equivalence checking of arithmetic circuits on the arithmetic bit level. IEEE Transactions on Computer Aided Design of Circuits and Systems **23**(5), 586–597 (2004)
27. Temel, M., Slobodová, A., Hunt, W.A.: Automated and scalable verification of integer multipliers. In: Computer Aided Verification. pp. 485–507 (2020)
28. Vasudevan, S., Viswanath, V., Sumners, R.W., Abraham, J.A.: Automatic verification of arithmetic circuits in RTL using stepwise refinement of term rewriting systems. IEEE Trans. on Comp. **56**(10), 1401–1414 (2007)
29. Wegener, I.: Branching Programs and Binary Decision Diagrams. SIAM (2000)
30. Wilhelm, U., Ebel, S., Weitzel, A.: Functional safety of driver assistance systems and ISO 26262. In: Handbook of Driver Assistance Systems: Basic Information, Components and Systems for Active Safety and Comfort, pp. 109–131. Springer (2016)
31. Wolf, C.: Yosys open synthesis suit. available at `http://www.clifford.at/yosys/` (2015)
32. Zimmermann, R.: Binary Adder Architectures for Cell-Based VLSI and their Synthesis. Ph.D. thesis, Swiss Federal Institute of Technology (1997)