Polynomial Formal Verification of Approximate Adders with Constant Cutwidth

Mohamed Nadeem
University of Bremen
Bremen, Germany
mnadeem@uni-bremen.de

Chandan Kumar Jha
University of Bremen
Bremen, Germany
chajha@uni-bremen.de

Rolf Drechsler
University of Bremen/DFKI
Bremen, Germany
drechsler@uni-bremen.de

***Abstract*—In the context of digital circuits, formal verification methods have been well-studied to ensure their functional correctness. However, several verification methods fail to provide an upper bound for the time and space complexity. Therefore, *Polynomial Formal Verification* (PFV) has been introduced to address this problem. Unlike prior works, which have shown that approximate circuits can be verified in polynomial time, we show that approximate circuits with a constant cutwidth can be verified even in linear time. Since approximate circuits have become ubiquitous in error-resilient applications, it becomes essential to guarantee their correctness. While prior works have been limited to formal error analysis, we use *Answer Set Programming* (ASP) based formal verification to guarantee that the approximate circuit matches its functional specification. In this paper, we first show that several approximate adder circuits exhibit a constant cutwidth. We then provide a PFV approach that relies on this cutwidth as a structural property of the circuits to guarantee a linear-time verification w.r.t. the bitwidth using ASP. Finally, we evaluate several approximate adders in terms of the upper bound of the cutwidth, and verification time.**

***Index Terms*—Polynomial Formal Verification, Logic Synthesis, Approximate Circuits, Dynamic Programming, Answer Set Programming, Cutwidth.**

I. Introduction

In the context of digital system design, ensuring the functional correctness of circuits is a crucial task. Circuit designs are becoming increasingly complex to cater to modern applications' demands, making the verification process more challenging. Thus, formal verification methods have been established to ensure that the circuit implementation matches its specifications [1], [2]. Several methods have been successfully employed for the formal verification of circuits, including *Boolean Satisfiability* (SAT), *Binary Decision Diagrams* (BDDs) [3], Symbolic Computer Algebra (SCA) [4], *Answer Set Programming* (ASP) [5]–[7], etc. While these methods are effective in ensuring functional correctness, the upper bound of the runtime for the verification process is still not guaranteed [8]. Therefore, *Polynomial Formal Verification* (PFV) [9] has been introduced to provide an upper bound of the time complexity.

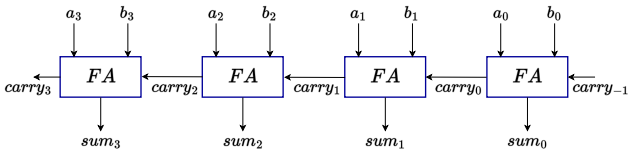
In prior works, it has been shown that several types of adders can be verified in polynomial time [10]–[12]. Recently, it has also been proven that adders with a constant *Cutwidth* [13] (i.e., *Ripple Carry Adder* (RCA), *Carry Look-ahead Adder* (CLA), and *Carry Skip Adder* (CSKA)) can be even verified in linear time [14]. In the context of formal verification, cutwidth corresponds to the minimum number of edge cuts required to split the circuit into subcircuits. However, these works have been limited only to exact adders.

In recent years approximate circuits have garnered significant attention as a result of the benefits obtained in power, performance, and area [15]–[17]. Error-resilient applications can produce an acceptable quality of output even after using approximate circuits for their computations. Formal verification methods for exact circuits cannot be directly used in approximate circuits as they do not have the same specifications. For example, functional approximation is one of the most popular methods to design an approximate circuit. In functional approximation, the exact Boolean function of the circuit is replaced with an approximate Boolean function [16], [17]. Hence, the specification for the approximate circuit differs from that of the exact circuit. As a result of this formal error analysis has been used for the verification of approximate circuits [18]. However, this method only guarantees that the approximate circuits' output error is below a specified error bound and provides no guarantee that it matches the functional specification. In another work, it was shown that approximate adders can be verified using BDDs in polynomial time [19].

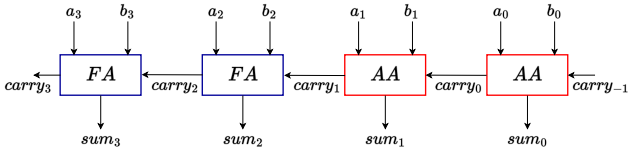
Prior works are limited as they either focus on performing formal error analysis or verifying the approximate adders in polynomial time. In this work, we alleviate these limitations and guarantee that *a) the approximate adders exactly matches its functional specification,* and *b) the approximate adders with a constant cutwidth can be formally verified in linear time*.

Our approach relies on the cutwidth to split the netlist into subcircuits. Thus, an ASP solver can verify each subcircuit independently, while the interconnected nodes (also called *Outgoing Nodes*) among the subcircuits are stored. Our paper represents a theoretical improvement over the state-of-the-art as we prove that the approximate adders with constant cutwidth can be verified in linear time. We also provide experimental validation to confirm our theoretical findings, using several approximate adders. These approximate adders can be divided into sub-adders, containing both exact and approximate sub-adders [20], [21]. These approximate sub-adders are obtained by changing or removing gates from the exact ones, i.e., using functional approximation [22]. We also experimentally evaluate more than 1.5 million approximate RCA of input bitwidth 8 and 16, to highlight the efficacy of the proposed approach. We do this as RCAs are mostly used in approximate adders [22], [23]. However, to show that our method is not only limited to RCA, we also show our proposed approach on approximate

This work was supported by the German Research Foundation (DFG) within the Reinhart Koselleck Project *PolyVer* (DR 287/36-1).



(a) Exact Adder



(b) Approximate Adder

Fig. 1: 4-bit Ripple Carry Adder

CLA and CSKA adders, as they also have a constant cutwidth and can be verified in linear time using ASP.

The paper is structured as follows: In Section II we introduce adder functions, approximate adder circuits, and cutwidth as a structural property of the *And-Inverter Graph* (AIG) [24] representation of a circuit, and the basic concepts of ASP. Subsequently, the modelling of circuits in ASP is described in Section III. Section IV presents our approach for PFV of approximate adder circuits. Section V describes the complexity properties of our approach. An experimental evaluation follows this in Section VI.

II. Preliminaries

A. Adder Function

Let a, b be two inputs with size n bits, and $carry_{-1}$ be the incoming carry bit. The adder function adds two inputs a_i and b_i together with $carry_{i-1}$, and outputs sum_i and $carry_i$, for all $0 \leq i \leq n$. The sum and carry functions can be characterized as follows.

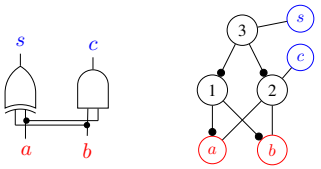
$$sum_i := a_i \oplus b_i \oplus carry_{i-1} \quad (1)$$

$$carry_i := (a_i \wedge b_i) \vee (carry_{i-1} \wedge (a_i \oplus b_i)) \quad (2)$$

Thus, the adder function takes $2n+1$ input bits that represent a_n, b_n and $carry_{-1}$, and $n+1$ output bits, where n bits represent sum and one carry output bit $carry_n$.

B. Approximate Adders

In this work, we have focused on functional approximation, i.e., replacing the Boolean function of the output sum_i and $carry_i$ with another function to obtain the approximate adders [22], [25]. The replaced Boolean function is chosen to obtain benefits in design metrics like power, performance, and area at the cost of producing inaccurate results for some input combinations [26]. The error-resilient applications produce acceptable quality outputs even after introducing these approximations [15]. The granularity of approximation can be varied by changing the number of exact adders that are being replaced with approximate adders [16], [22]. The example of a 4-bit exact and 4-bit approximate RCA with two approximated bits is shown in Fig. 1. In prior works, it has been shown that the approximate Boolean function is carefully crafted for a particular application [17]. Thus, the implementation of an approximate adder should exactly match its specification.



(a) HA (b) AIG

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(c) Truth Table

Fig. 2: The logic diagram of a Half Adder is shown in Fig. 2(a), its corresponding AIG graph illustrated in Fig. 2(b) and its truth table outlined in Fig. 2(c).

C. Cutwidth of AIG

Informally, a netlist on the reverse topological order can be represented as a directed acyclic graph *AIG* G, which is composed of the sets of terminal nodes representing the inputs PI and the outputs PO, and non-terminal nodes representing the *And* gates and the *Inv* gates. This can be formulated as follows.

Definition 1 (AIG Graph): Let $G = (V, E)$ be an *AIG* of a netlist C such that:

- $V := \{v \mid v \text{ is a node}\}$.
- $E := \{(v, v') \mid v, v' \in V, v \text{ is connected to } v'\}$.

Given the block diagram in Fig. 2(a), the AIG graph G is constructed as shown in Fig. 2(b) and the truth table is shown in Fig. 2(c).

A cutwidth of a graph G of a linear ordering $v_1, ..., v_n$ is the smallest integer k such that for every $i = 1, ..., n-1$, there exists at most k edges with one endpoint $v_1, ..., v_i$ and the other in $v_{i+1}, ..., v_n$. We refer by CO (called *Out-going Nodes*) to the set of nodes induced by the *Edge-Cuts* to partition the graph G into a set of subgraphs $G_1, ..., G_n$. This yields a characterization of *K-bounded Graph* that is defined in [27] as follows.

Definition 2 (K-bounded Graph): Let K be a positive number. Then, a graph G is said to be *K-bounded* if there exists a partition $\sigma = \{G_1, ..., G_n\}$ of G such that for every $G_i \in \sigma$, the number of inputs is at most K.

D. Answer Set Programming

ASP is a well-known declarative programming framework from the area of knowledge representation and non-monotonic reasoning [28]. It is mainly used to solve NP-hard search problems, while allowing a compact modelling [29], and the search problems are reduced for computing *Answer Sets*. We follow the standard definitions of propositional ASP [30] (for more details, see [31]).

Let $a_1, ..., a_n$ be distinct *Atoms*. Then, a program Π is defined in terms of rules over $a_1, ..., a_n$ as follows.

Definition 3 (Logic program): Let $a_1, ..., a_n$ be distinct atoms, and l, m, n be non-negative integers such that $l \leq m \leq n$. A *Logic Program* Π is a finite set of *Rules* of the form $a_1 \vee ... \vee a_l \leftarrow a_{l+1}, ..., a_m, \neg a_{m+1}, ..., \neg a_n$.
We denote by $Head(r)$, $Body(r)$ to the set of atoms appearing on the left-hand side and right-hand side of the rule r, respectively. A rule r is said to be a *Fact* (*Negation-free*) if $Body(r) = \emptyset$. In the context of circuit design, a circuit is modelled as a program Π. Then, a query Q (represented by a set of facts) representing values of circuit inputs, is added to the program Π (denoted by Π^Q). Thus, an ASP solver is asked, whether there exists an answer set of Π^Q. The answer set is

defined in terms of the *Gelfond-Lifschitz* (GL) *Reduct* [32] as follows.

Definition 4 (Answer Set): Given an interpretation I (represented by a set of atoms), and a program Π^Q. Then, I is an answer set of Π^Q iff I is a minimal model of Π^Q.
We refer by $AS(\Pi)$ to the set of all answer sets of the program Π w.r.t. all possible queries Q. To illustrate the previous definition, the answer set corresponds to the minimal set of atoms, satisfying Π^Q. It is important to distinguish the ASP semantics from the classical logic. In ASP, an answer set is a model of a program Π^Q, while a model is not required to be minimal in classical logic.

Example 1: Consider the program of the circuit in Fig. 2(a):

$$\Pi := \{s \leftarrow a, \neg b; s \leftarrow \neg a, b; \top \leftarrow s, a, b; \top \leftarrow \neg a, \neg b; \\ c \leftarrow a, b; \}$$

The sum (*Xor*) is captured by the first four rules. The first two rules illustrate that to set the sum s to 1, whenever a and b are different. Similarly, the second two rules describe that s is equal to 0, whenever a and b are the same. The last rule is defined similarly to encode the carry c (*And*). Now, the possible values for a and b are added as queries. This can be described as $Q = \{a \vee \neg a \leftarrow; b \vee \neg b \leftarrow\}$. Hence, $AS(\Pi) := \{\{a, s\}, \{b, s\}, \{a, b, c\}\}$ w.r.t. Π^Q.

III. Circuit modelling Using ASP

In this section, we illustrate the modeling of the AIG graph into ASP, where we rely on the input language of the ASP solver *Clingo* [33]. Clingo provides an interface for logical *Or* (as "X?Y"), *And* (as "X&Y"), *Xor* (as "X^Y"), and *Inv* (as "1^X"). Also, it uses ":-" to represent the left arrow symbol. Moreover, it utilizes uppercase and lowercase letters to denote variables and constants, respectively. The general idea is to introduce ASP rules that represent PI, PO, *And* and *Inv* gates, a connection between two arbitrary gate ports, and the circuit specification (exact and approximate adder functions). The actual representation of AIG (type of a gate, and connection between two different gate ports), and the values of PI (representing an ASP query) are encoded as ASP facts. Then, the ASP solver is used to reason whether each output $o \in PO$ matches its corresponding specification.

Each gate behavior is defined based on values on their ports, where these ports provide an interface for passing values between a gate and its connections. Let $P(G)$ be a unary function symbol representing a port of gate G, and $val(P(G), v)$ be a binary predicate symbol stating a value v on a port P of gate G. Also, let $conn(P1, P2)$ be a binary predicate symbol, representing the connection between ports $P1$ and $P2$. Furthermore, let $type(G, T)$ be a binary predicate symbol that is used to label a gate G with a type T (i.e., *And*, *Inv*, PI, and PO). Finally, the rules representing *And*, *Inv*, and the connection between two ports are characterized as follows.

$$val(out(G), X\&Y) \text{:- } type(G, and), \\ val(in1(G), X), val(in2(G), Y). \quad (3)$$

$$val(out(G), 1\hat{}X) \text{:- } type(G, inverter), val(in(G), X). \quad (4)$$

$$val(P2, V) \text{:- } conn(P1, P2), val(P1, V). \quad (5)$$

In Eq. (3), the *And* gate behavior is captured such that X and Y indicate the values on the ports $in1(G)$ and $in2(G)$ of gate G, respectively, while the value obtained from performing logical *And* is passed to the port $out(G)$. Similarly, Eq. (4) captures the *Inv* gate such that it takes value X on port $in(G)$ of gate G and performs the logical *Inv* and passes it to $out(G)$. Finally, Eq. (5) captures passing the value V from port $P1$ to port $P2$.

To complete the modeling of the circuit, we introduce facts representing the actual representation of AIG. Considering Fig. 2(b), the facts $type(and2, and)$, $conn(b, in1(and2))$, and $conn(a, in2(and2))$ capture the *And* gate "2", as well as its connection with inputs a and b. Furthermore, it is essential to encode the circuit specification. Thus, the output gate can be checked against its corresponding logic function. The exact sum and carry rules (recall Eq. (1) and Eq. (2)) are formulated as follows.

$$sum(sum_i, V) \text{:- } val(a_i, A), val(b_i, B), \\ carry(carry_{i-1}, C), V = A\hat{}B\hat{}C. \quad (6)$$

$$carry(carry_i, V) \text{:- } val(a_i, A), val(b_i, B), \\ carry(carry_{i-1}, C), V = (A\&B)?(C\&(A\hat{}B)). \quad (7)$$

Since there exist several ways to approximate the sum and the carry functions, they can be encoded similarly using the Clingo logical representation of the logic functions (e.g., *Or* and *Xor*). Also, it is essential to add a rule that relates each output gate with its expected logic function to enable verification of the circuit. This is characterized as follows.

$$verify(o_i) \text{:- } sum(sum_i, X), val(o_i, X). \\ verify(o_n) \text{:- } carry(carry_n, X), val(o_n, X).$$

Finally, the values PI are encoded as facts (i.e., $val(a, 0)$ and $val(b, 0)$ correspond to the first row of the table illustrated in Fig. 2(c)) that represent the ASP query. Let $\Pi(G)$ be the resulting ASP program. The set of values is said to be a *Valid Input* if every output gate matches its corresponding logic function (every $verify(o_i)$ appears in the answer set of $\Pi(G)$). This yields a characterization for a verification of the graph.

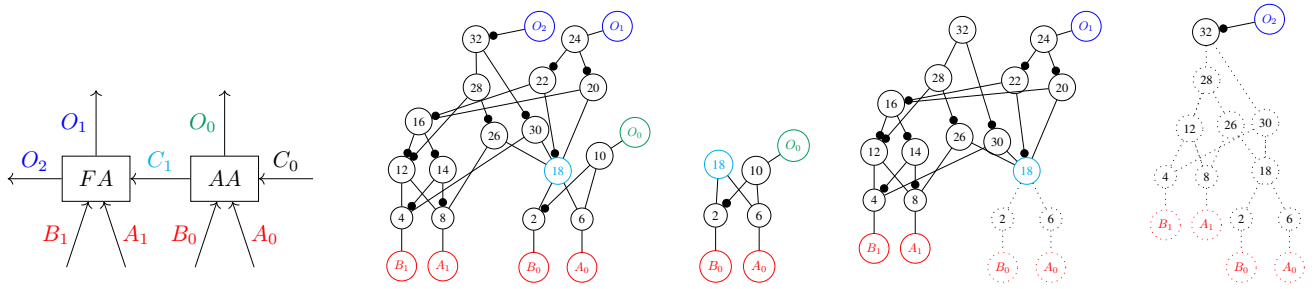
Definition 5 (Valid Graph): Let $\Pi(G)$ be a program defined w.r.t. the AIG graph G of size n, and $\mathcal{F}$ be a set of sets of facts such that each $s \in \mathcal{F}$ represents an input sequence. Then, G is said to be a *Valid Graph*, iff for every $s \in \mathcal{F}$, we have that s is a valid input. Otherwise, G is an *Invalid Graph*.
It is worth noting that the overall search space is 2^n. Hence, $|\mathcal{F}| = 2^n$. The following section illustrates the PFV approach for defining an upper bound of the search space.

IV. Polynomial Formal Verification of Approximate Adder Circuits

In this section, we extend the PFV approach of adder circuits illustrated in [14] to the case of approximate adders. We rely on the cutwidth property to split the AIG G into subgraphs, where the interleaving nodes (out-going nodes) are stored to be used in other subgraphs. Thus, we provide a method for passing the out-going nodes over subgraphs. Finally, we illustrate the verification of the subgraphs.

A. Graph Splitting

An AIG graph G can be split into subgraphs such that each subgraph starts from one output node and traverses all nodes,



(a) 2-bit RCA block diagram (b) AIG G (c) (G, O_0). (d) (G, O_1). (e) (G, O_2).

Fig. 3: The AIG G of 2-bit approximate RCA block diagram in Fig. 3(a) and the resulting (reduced) subgraphs (G, O_0), (G, O_1) and (G, O_2) that can be obtained from the AIG graph in Fig. 3(b). The nodes highlighted in red and blue correspond to input and output nodes, respectively. The nodes highlighted in cyan and green correspond to carry C_1 and approximate output O_0, respectively. Dotted nodes and edges are removed when the subgraph is reduced.

from which the output node is reachable. This is characterized as follows.

Definition 6 (Subgraph): Let $G = (V, E)$ be a graph, and $v \in V$ be an output node. Then, a subgraph $(G, v) = (V_v, E_v)$ of G is obtained such that:

- $V_v := \{v\} \cup \{v' \in V \mid v'$ is reachable from $v\} \cup \{v' \in V \mid \exists x, y \in V : x, y$ are reachable from $v', v\}$.
- $E_v := \{(u', v') \in E \mid u', v' \in V_v\}$.

Consider the graph G of Fig. 3(b). It can be split into subgraphs (G, O_0), (G, O_1), and (G, O_2) as shown in Fig. 3(c), Fig. 3(d), and Fig. 3(e), respectively. It is clear to see that some nodes (out-going nodes) appear in several subgraphs (e.g., node "18"). We refer by G_i to the *Reduced Subgraph* of (G, O_i) that is obtained from removing all nodes and edges that appear in any subgraph (G, O_j), where $j < i$. This set of nodes is defined w.r.t. G_i as follows.

Definition 7 (Out-going Nodes): Let $G = (V, E)$ be AIG graph, and $G_i = (V_i, E_i)$ be the reduced subgraph of (G, O_i), where $0 \leq i \leq n$. The set CO_i of *Out-going Nodes* is defined w.r.t. G_i such that $CO_i := \{a \in V_i \mid (b, a) \in E, b \notin V_i\}$. Similarly, we refer by CI_i to the set of *In-going Nodes* containing all nodes with a predecessor node that appears in any other graphs G_j, where $j < i$. For the graph G_1 in Fig. 3(d), we have that $CO_1 = \{32\}$, and $CI_1 = \{18\}$. Notably, $CI_0 = \emptyset$ of G_0, while $CO_n = \emptyset$ of G_n. As the reduced subgraph G_i may contain primary inputs PI_i, and non-primary inputs (in-going nodes) CI_i, we denote by IN_i to the set of all inputs of the reduced subgraph G_i.

B. Information Passing

Intuitively, the set of out-going nodes CO_i is evaluated in the reduced subgraph G_i. Thus, the values of CO_i need to be stored to be used in any other subgraphs G_j, where $j > i$. Notably, these values cannot be stored as a function of the primary inputs, as this would require passing the primary inputs to other subgraphs. Instead, the values of CO_i are stored w.r.t. the computed carry function. This allows us to use the carry function $carry_i$ in the specification of the output for the reduced subgraph. Therefore, we define two mapping functions f and g as follows.

$$f\colon IN_i \mapsto COUT_i. \tag{8}$$

$$g\colon COUT_i \mapsto [0, 1]. \tag{9}$$

Intuitively, function f maps each set of input values $s \in IN_i$ to a set of values $COUT_i$ of out-going nodes CO_i. Then, function g maps the resulting values $s' \in COUT_i$ to the value of carry function $carry_i$. Finally, a hash table $\mathcal{X}_i$ is used to store the resulting values as follows.

$$\mathcal{X}_i = \{(f(s), g(f(s))) \mid s \in IN_i\}. \tag{10}$$

To illustrate this, consider Fig. 3(c), and suppose $carry_{-1} = 0$. Then, $CO_0 = \{18\}$, and $CI_0 = \emptyset$. There are two possible values of f, i.e., $s \in COUT_0$: $f(0, 0) = \{0\}$ and $f(1, 1) = \{1\}$. Other input combinations do not need to be considered, as they are already covered in $COUT_0$. Notably, the hash table $\mathcal{X}_i$ is bounded by the size of CO_i (e.g., $\mathcal{X}_0 = \{(0, 0), (1, 1)\}$).

C. Subgraph Verification

The verification of G_i involves checking whether G_i is valid (recall Definition 5), and constructing $\mathcal{X}_i$ to be used in CI_j of subgraph G_j, where $j > i$. This is because inputs IN_i of G_i may contain primary inputs PI_i and in-going nodes CI_i. As CI_i may be stored in any hash table $\mathcal{X}_j$, where $j < i$, it is essential to go over all tables $\mathcal{X}_j$ to obtain all values of CI_i. A relation $\bowtie$ is used to define the relation between two table $\mathcal{X}_j$ and $\mathcal{X}_{j'}$ such that $\mathcal{X}_j \bowtie \mathcal{X}_{j'} := \{r \cup r' \mid r \in \mathcal{X}_j, r' \in \mathcal{X}_{j'}, CO_j \cap CO_{j'} \subseteq CI_i\}$. The values of the tables $\mathcal{X}_j$ and $\mathcal{X}_{j'}$ are combined, if they have out-going nodes that appear in the in-going nodes C_i of subgraph G_i, where $j, j' < i$. The resulting table $\mathcal{X}_i(CI_i)$ is defined as follows.

$$\mathcal{X}_i(CI_i) := \mathcal{X}_{i-1} \bowtie \ldots \bowtie \mathcal{X}_0. \tag{11}$$

Hence, the resulting table $\mathcal{X}_i(CI_i)$ is populated with the values of PI_i. Consider Fig. 3(d), in which $CI_1 = \{18\}$ is populated with $PI_1 = \{A_1, B_1\}$, where $\mathcal{X}_0(CI_0) = \{(0, 0), (1, 1)\}$.

V. Time Complexity

In this section, we illustrate that the upper bound of the cutwidth property relies on the exact sub-adder and is independent of the approximate sub-adder. Earlier, it has been proven in [14] that the program $\Pi(G_i)$ of the reduced subgraph G_i is verified in $\mathcal{O}(2^{|IN_i|})$. Also, it has been proven that the overall graph G with a constant cutwidth cw is verified in time $\mathcal{O}(n \cdot 2^K)$, where n is the number of subgraphs, and $K = max(|IN_0|, ..., |IN_n|)$.

Lemma 5.1: Let A be an exact adder of size n. If the adder A has a constant cutwidth cw and can be verified in $\mathcal{O}(n \cdot 2^K)$, then an approximate adder A' constructed w.r.t. A with x approximate bits also exploits a constant cutwidth cw' and can be verified in $\mathcal{O}(n \cdot 2^{K'})$ such that $cw' \leq cw$ and $K' \leq K$.

TABLE I: Calculated cutwidth (cw), upper bound (K), number of approximate bits (#Approx), clingo solving, clingo modelling, and verification time (seconds) for the approximate adders of size 8.

#Approx	cw	K	AVG #Gates of Graph				AVG #Gates of Subgraphs				Solving Time[s]			Modelling Time[s]			Verification Time[s]		
			#Inputs	#Outputs	#Ands	#Gates	#Inputs	#Outputs	#Ands	#Gates	Min	Max	Mean	Min	Max	Mean	Min	Max	Mean
1	1	3	16	9	74.6	104.6	3	2	10.0	15.0	0.03	0.06	0.03	0.02	0.03	0.02	0.42	0.72	0.51
2	1	3	16	9	79.6	104.5	3	2	15.5	19.6	0.02	0.06	0.03	0.01	0.03	0.02	0.42	0.75	0.52
3	1	3	16	9	84.6	109.5	3	2	15.5	19.6	0.02	0.06	0.03	0.01	0.03	0.02	0.42	0.71	0.52
4	1	3	16	9	89.6	114.5	3	2	15.5	19.6	0.02	0.06	0.03	0.01	0.03	0.02	0.42	0.73	0.52
5	1	3	16	9	94.6	119.5	3	2	15.5	19.6	0.02	0.07	0.03	0.01	0.03	0.02	0.42	0.73	0.52
6	1	3	16	9	99.6	124.5	3	2	15.5	19.6	0.02	0.06	0.03	0.01	0.03	0.02	0.42	0.70	0.52
7	1	3	16	9	104.6	129.5	3	2	15.1	19.2	0.02	0.07	0.03	0.02	0.03	0.02	0.42	0.73	0.52

TABLE II: Calculated cutwidth (cw), upper bound (K), number of approximate bits (#Approx), clingo solving, clingo modelling, and verification time (seconds) for the approximate adders of size 16.

#Approx	cw	K	AVG #Gates of Graph				AVG #Gates of Subgraphs				Solving Time[s]			Modelling Time[s]			Verification Time[s]		
			#Inputs	#Outputs	#Ands	#Gates	#Inputs	#Outputs	#Ands	#Gates	Min	Max	Mean	Min	Max	Mean	Min	Max	Mean
1	1	3	32	17	154.6	203.5	3	2	10.0	15.0	0.06	0.96	0.06	0.03	0.54	0.03	0.42	9.02	0.60
2	1	3	32	17	159.6	208.5	3	2	15.5	19.6	0.05	0.98	0.06	0.03	0.55	0.03	0.42	9.03	0.60
3	1	3	32	17	164.6	213.5	3	2	15.5	19.6	0.05	0.97	0.06	0.03	0.54	0.03	0.43	9.00	0.60
4	1	3	32	17	169.6	218.5	3	2	15.5	19.6	0.05	1.02	0.06	0.03	0.55	0.03	0.43	9.10	0.60
5	1	3	32	17	174.6	223.5	3	2	15.5	19.6	0.05	1.04	0.06	0.03	0.56	0.03	0.42	9.10	0.60
6	1	3	32	17	179.6	228.5	3	2	15.5	19.6	0.04	1.03	0.06	0.03	0.55	0.03	0.42	9.11	0.83
7	1	3	32	17	184.6	233.5	3	2	15.5	19.6	0.04	2.74	0.06	0.03	0.55	0.03	0.42	9.06	0.60
8	1	3	32	17	189.6	238.4	3	2	15.5	19.6	0.04	1.08	0.06	0.03	0.56	0.03	0.42	9.19	0.60
9	1	3	32	17	194.6	243.4	3	2	15.5	19.6	0.04	1.16	0.06	0.03	0.57	0.03	0.32	9.21	0.60
10	1	3	32	17	199.6	248.4	3	2	15.5	19.6	0.04	1.21	0.06	0.02	0.58	0.03	0.42	9.32	0.60
11	1	3	32	17	204.6	253.4	3	2	15.5	19.6	0.03	1.22	0.06	0.02	0.59	0.03	0.42	9.41	0.60
12	1	3	32	17	209.6	258.4	3	2	15.5	19.6	0.03	1.18	0.07	0.02	0.58	0.03	0.42	9.33	0.60
13	1	3	32	17	214.6	263.4	3	2	15.5	19.6	0.03	1.30	0.07	0.02	0.58	0.03	0.43	9.46	0.61
14	1	3	32	17	219.6	268.3	3	2	15.5	19.6	0.03	1.19	0.07	0.02	0.57	0.03	0.42	9.28	0.61
15	1	3	32	17	224.6	273.3	3	2	15.1	19.2	0.03	1.11	0.07	0.03	0.58	0.04	0.42	9.20	0.61

Proof: Let A be an exact adder circuit of size n with a constant cutwidth cw. Then, the AIG graph G constructed w.r.t. A can be split into $(G, O_0), ..., (G, O_n)$ by starting from any output node O_i and traversing all nodes that are reachable from O_i, where $0 \leq i \leq n$. Moreover, let $G_0, ..., G_n$ be the reduced subgraphs of $(G, O_0), ..., (G, O_n)$. Now let A' be an approximate adder of A with x approximate bits. Similarly, let G' be the AIG graph of A'. Also, let $(G', O_0), ..., (G', O_n)$ be the resulting subgraphs of G'. Consequently, let $G'_0, ..., G'_n$ be the resulting reduced subgraphs. As A' has x approximate bits, let $G'_0, ..., G'_x$ be the approximate reduced subgraphs such that $x \leq n$ with inputs $IN'_0, ..., IN'_x$.

As the approximate adder A' is derived from A by either removing gates or replacing some gates with others, and the approximation is performed at the sub-adder level (bit level), the number of edges connecting any two sub-adders either remains constant, in the case of retaining the same number of nodes representing the carry or decreases due to the approximation of the carry. Therefore, for every CO'_i of G'_i and CO_i of G_i, we have that $|CO'_i| \leq |CO_i|$, where $0 \leq i \leq x$. Let cw be the cutwidth of the adder A such that $cw = max(CO_0, ..., CO_n)$. Similarly, let cw' be the cutwidth of the adder A' such that $cw = max(CO'_0, ..., CO'_n)$. Then, $cw' \leq cw$. As A has a constant cutwidth cw, then $\forall i, 0 \leq i \leq n : |CO_i| \leq cw$. Hence, $cw' \leq cw$.

Also, for every IN'_i, IN_i we have that $|IN'_i| \leq |IN_i|$, where $0 \leq i \leq x$. Let K be the upper bound of inputs of the adder A such that $K = max(IN_0, ..., IN_n)$. Similarly, let K' be the upper bound of inputs of the adder A' such that $K = max(IN'_0, ..., IN'_n)$. Hence, $K' \leq K$.

Therefore, if the exact adder A has a constant cutwidth cw and the approximate adder A' of A, then, there exists an integer $cw' \leq cw$ such that cw' is the cutwidth of the approximate adder A'. Also, if the adder A can be verified in $\mathcal{O}(n \cdot 2^K)$, then the approximate adder A' can be verified in $\mathcal{O}(n \cdot 2^{K'})$. ■

VI. Experimental Work

To check the feasibility of our approach, we have implemented the ASP framework in Python. The framework takes an input approximate circuit in the standard AIGER format [34], and circuit specification in the Verilog. We evaluate approximate RCA, CSKA, and CLA of different input bitwidth under different functional approximations, as it has been shown in [14] that the exact circuits of RCA, CSKA, and CLA have a constant cutwidth. The approximate RCA is designed by generating all possible approximations for the two outputs. Since each of the two outputs is dependent on 3 inputs, the number of possible functions that can be generated is $2^{2^3} * 2^{2^3}$, i.e., 65,536 designs for each approximated bit [22]. The CSKA and CLA adders are generated using the ArithsGen tool [35]. These adders of size n are split into blocks of size 4. For CSKA and CLA adders, we have chosen one approximation and the number of approximated bits is 4, to highlight that our method is not only limited to RCA. All experiments were performed on an Intel(R) Core(TM) i7-11370 with 3.30 GHz. We set a timeout of 100 seconds and a limited available RAM to 16 GB per instance.

A. Approximate RCA

We evaluate 1.5 million approximate RCA with 8 and 16 input bitwidth with different numbers of approximate bits up to 7 and 15, respectively in terms of the cutwidth, the upper

TABLE III: Calculated upper bound (K), and verification time (VT) (seconds) w.r.t. size (n) for CSKA and CLA, where #Approx= 4.

n	CSKA		CLA	
	K	VT	K	VT
8	8	0.51	11	0.53
16	8	0.61	11	0.72
32	8	0.81	11	1.09
64	8	1.29	11	1.85
128	8	2.24	11	3.36

bound for the upper bound of inputs, verification time, clingo modeling time, and clingo solving time.

Table I presents the results for approximate RCA adders of size 8 ($n = 8$) with all possible functional approximations. It shows the number of approximate bits (#Approx), the cutwidth (the maximum number of out-going nodes) cw, the upper bound for inputs (the maximum number of in-going nodes) K, the number of instances, the average number of gates in the input graph, and the average number of gates in reduced subgraphs. Moreover, it provides the minimum, maximum, and average times in seconds for clingo modeling, clingo solving, and verification. As $cw = 1$ for different functional approximations and different numbers of approximate bits, this aligns with Lemma 5.1 that the cutwidth is bounded by the regular sub-adders and does not rely on the approximate sub-adders.

Similarly, Table II presents the results for approximate RCA adders of size 16 ($n = 16$) with all possible functional approximations. Since the upper bound K is equal to 3 (i.e., two primary inputs PI and one in-going node CI representing the incoming carry) for $n = 8$, and $n = 16$, this confirms the linearly time scalability that has been shown in [14].

B. Approximate CLA and CSKA

To show the linear time scalability of the verification process, we evaluate approximate CLA and CSKA with different input bitwidth up to 256 in terms of the upper bound K and verification time, where the number of approximate bits is equal to 4 and the functional approximation is fixed.

Table III shows the upper bound for the inputs K, and verification time w.r.t. approximate CLA and CSKA. It shows that both approximate circuits exhibit a constant cutwidth. Therefore, the verification process scales in linear time w.r.t. the input bitwidth. Also, it confirms Lemma 5.1 showing that the upper bound K is bounded by the regular sub-adders and does not rely on the approximate sub-adders.

VII. Conclusion

In this paper, we have proven that approximate adders with a constant cutwidth can be verified in linear time w.r.t. the input bitwidth. These linear time verifiable approximate adders are generated from RCA, CSKA, and CLA regular adders by performing functional approximation. Also, we have shown that the upper bound for the cutwidth depends on the exact sub-adders and does not depend on the approximate ones. Finally, the experimental evaluations confirm the upper bound complexity of each approximate adder architecture.

References

[1] R. Drechsler, Ed., *Advanced Formal Verification*. Kluwer Academic Publishers, 2004.

[2] R. Drechsler, *Formal System Verification: State-of the-Art and Future Trends*, 1st ed. Springer Publishing Company, Incorporated, 2017.

[3] R. Drechsler *et al.*, "Binary decision diagrams in theory and practice," *STTT*, vol. 3, pp. 112–136, 2001.

[4] A. Mahzoon *et al.*, "Revsca-2.0: Sca-based formal verification of nontrivial multipliers using reverse engineering and local vanishing removal," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 5, pp. 1573–1586, 2021.

[5] G. Brewka *et al.*, "Answer set programming at a glance," *ACM*, vol. 54, no. 12, p. 92–103, 2011.

[6] M. Gebser *et al.*, *Answer Set Solving in Practice*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning, 2012.

[7] A. Provetti *et al.*, "Answer set programming: Towards efficient and scalable knowledge representation and reasoning," in *Proceedings of the 1st Intl. ASP'01 Workshop*, 2001.

[8] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*. ACM, 1971, pp. 151–158.

[9] R. Drechsler *et al.*, "Polynomial formal verification: Ensuring correctness under resource constraints," in *ICCAD*, 2022, pp. 70:1–70:9.

[10] R. Drechsler, "PolyAdd: Polynomial formal verification of adder circuits," in *DDECS*, 2021, pp. 99–104.

[11] A. Mahzoon *et al.*, "Late breaking results: Polynomial formal verification of fast adders," in *DAC*, 2021, pp. 1376–1377.

[12] A. Mahzoon *et al.*, "Polynomial formal verification of prefix adders," in *ATS*, 2021, pp. 85–90.

[13] F. R. K. Chung, "On the cutwidth and the topological bandwidth of a tree," *SIDMA*, vol. 6, no. 2, pp. 268–277, 1985.

[14] M. Nadeem *et al.*, "Polynomial formal verification exploiting constant cutwidth," in *Proceedings of the 34th International Workshop on Rapid System Prototyping*. IEEE, 2023.

[15] W. Liu *et al.*, "A retrospective and prospective view of approximate computing," *Proceedings of the IEEE*, vol. 108, no. 3, pp. 394–399, 2020.

[16] C. K. Jha *et al.*, "Energy and error analysis framework for approximate computing in mobile applications," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 2, pp. 385–389, 2020.

[17] C. K. Jha *et al.*, "Single exact single approximate adders and single exact dual approximate adders," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 7, pp. 907–916, 2023.

[18] Z. Vasicek, "Formal methods for exact analysis of approximate circuits," *IEEE Access*, vol. 7, pp. 177 309–177 331, 2019.

[19] M. Schnieber *et al.*, "Polynomial formal verification of approximate adders," in *2022 25th Euromicro Conference on Digital System Design (DSD)*, 2022.

[20] S. Froehlich *et al.*, "Approximate hardware generation using symbolic computer algebra employing Gröbner basis," in *DATE*, 2018, pp. 889–892.

[21] R. Hrbacek *et al.*, "Automatic design of approximate circuits by means of multi-objective evolutionary algorithms," in *2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2016, pp. 1–6.

[22] C. K. Jha *et al.*, "Imagin: Library of imply and magic nor-based approximate adders for in-memory computing," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 8, no. 2, pp. 68–76, 2022.

[23] V. Mrazek *et al.*, "Evoapproxlib: Extended library of approximate arithmetic circuits," in *Proc. Workshop Open-Source EDA Technol.(WOSET)*, 2019, p. 10.

[24] A. Mishchenko *et al.*, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *DAC*, 2006, pp. 532–535.

[25] V. Gupta *et al.*, "Low-power digital signal processing using approximate adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, 2012.

[26] R. Venkatesan *et al.*, "Macaco: Modeling and analysis of circuits for approximate computing," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2011, pp. 667–673.

[27] H. Fujiwara, "Computational complexity of controllability/observability problems for combinational circuits," *IEEE Trans. Computers*, vol. 39, no. 6, pp. 762–767, 1990.

[28] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[29] M. Gebser *et al.*, "Conflict-driven answer set solving: From theory to practice," *Artificial Intelligence*, 2012.

[30] "Logic programs with stable model semantics as a constraint programming paradigm," *Annals of Mathematics and Artificial Intelligence*, 1999.

[31] V. W. Marek *et al.*, "Stable models and an alternative logic programming paradigm," *A Computing Research Repository*, 1998.

[32] M. Gelfond *et al.*, "Classical negation in logic programs and disjunctive databases," *New Generation Computing*, pp. 365–385, 1991.

[33] M. Gebser *et al.*, "Advances in gringo series 3," in *LPNMR*, 2011, pp. 345–351.

[34] A. Biere, "The AIGER And-Inverter Graph (AIG) format version 20071012," Institute for Formal Models and Verification, Johannes Kepler University, Tech. Rep., 2007.

[35] J. Klhufek *et al.*, "Arithsgen: Arithmetic circuit generator for hardware accelerators," in *DDECS*, 2022, pp. 44–47.