Polynomial Formal Verification exploiting Constant Cutwidth

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

Jan Kleinekathöfer
University of Bremen
Bremen, Germany
ja_kl@uni-bremen.de

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

***Abstract*—Only formal methods can guarantee the correctness of a circuit, but are usually very time and memory consuming. Therefore, efficient formal verification is a key in the design of complex circuits. Many verification techniques have been introduced, which mostly fail to give bounds for the time complexity of the verification process. To overcome this issue, *Polynomial Formal Verification* (PFV) was introduced. This paper introduces a novel approach to PFV of circuits, by leveraging the concept of constant cutwidth. We divide the circuit into subgraphs, one for every output.This makes the verification of every subgraph only dependent on the cutwidth of the circuit and independent of the bitwidth. One main problem we solve is the passing of information between those subgraphs. The approach enables formal verification in linear time for circuits with constant cutwidth.**

As many different types of adders have a constant cutwidth, we can prove that those are verifiable in linear time. Those theoretical findings are backed by experiments including different adder architectures with up to 10k bit wide inputs.

***Index Terms*—Polynomial Formal Verification, Logic Synthesis, Parameterized Complexity, Answer Set Programming, Model Based Reasoning**

I. Introduction

Circuit designs are constantly getting more complex due to advancements in process technology. Moreover, in recent time more custom designs are implemented to increase the performance for regularly occurring computation tasks (i.e. for AI or AES). To guarantee the correctness of those circuits formal methods are needed. Due to increasing complexity, the scalability of the verification process is highly relevant.

Various techniques have been introduced for the verification of circuits, such as SAT-based approaches that encode verification as a SAT instance using miter circuits [1], decision diagrams that represent the output function of circuits for comparison to the specification [2], theorem proving for manual verification of complex circuits [3], *Symbolic Computer Algebra* (SCA) that is effective for multiplier verification [4], and *Answer Set Programming* (ASP) that provides a compact representation of logic functions [5]–[7]. However, the computational complexity of these methods is often exponential, such as with the NP-hard SAT problem, the exponentially large decision diagrams, and the exponential size of polynomials in SCA. To address these limitations, *Polynomial Formal Verification* (PFV) [8], [9] methods have been introduced to guarantee a polynomial run time of the formal verification for a specific class of circuits.

In this paper, our investigation is centered around circuits characterized by a limited cutwidth [10], [11]. Cutwidth in the context of formal verification denotes the minimum number of layers that allow each wire to intersect at most one layer. It simplifies the analysis and verification process, making it easier to ensure the correctness of circuit designs. However, it is important to note that not all circuits exhibit a constant cutwidth. Therefore, our research focuses on the implications and advantages specific to circuits with this structural property. Recent progress in efficient cutwidth computation algorithms [12] for large graphs have underscored the growing relevance of cutwidth for verifying complex designs.

Our paper presents a new method for verifying the specific class of circuits with a constant cutwidth. The proposed approach involves dividing the netlist into subcircuits, which are independently verified by an ASP solver. The subcircuits are interconnected through cone nodes, employing a divide and conquer strategy. Our results demonstrate that the proposed method verifies circuits with a constant cutwidth in linear time.

This is particularly significant as the complexity of the verification is mainly determined by the cutwidth between the subcircuits. Moreover it aligns with previous research on automatic test pattern generation [13]. The approach uses ASP to model the subproblems, enabling a compact and straightforward approach to verification. While various techniques, such as SAT or SMT solvers, can be used in conjunction with other preprocessing approaches, the primary focus of this paper is on introducing the proposed approach in combination with ASP. Our work presents a novel approach in the field of PFV, being the first to utilize ASP in this domain.

Our theoretical results for adders improve the state-of-the-art. While it is known from practice, that adders can be verified efficiently, the time complexity results are limited to [8], [14], [15]. With BDDs adders can be verified in polynomial time. *Conditional Sum Adders* (CSAs) can be verified in $O(n^4)$. Serial prefix adders, *Ladner-Fisher Adders* (LFAs) and *Kogge-Stone Adders* (KSAs) can be verified in $\mathcal{O}(n^2)$, $\mathcal{O}(n^4)$ and $\mathcal{O}(n^4)$, respectively. We extend this list by proving that *Carry SKip Adders* (CSKAs) and *Carry Look-ahead Adders* (CLAs) can be verified in linear time using our approach. The experiments demonstrate the effectiveness of the proposed approach, examining its performance for different input sizes and architectures. Moreover, we compare our approach against

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

Yosys [16] SAT-based approach, and show the feasibility of our approach over the SAT-based one.

In Section II we introduce the adder functions as well as the relevant the concept of ASP. Subsequently the modeling of circuits in ASP is described in Section III. Our approach for PFV by splitting the circuit is presented in Section IV. Section V describes the complexity properties of our approach. This is followed by an experimental evaluation 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 its output are the sum sum_i and $carry_i$, for all $0 \leq i \leq n$. The sum bits can be characterized as follows.

$$sum_i := a_i \oplus b_i \oplus carry_{i-1} \tag{1}$$

The carry bits can be characterized as follows.

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

Thus, the adder function has $2n+1$ input bits, and $n+1$ output bits. This is due to the fact that it adds two n bit inputs together with $carry_{-1}$, while it results in n bits representing sum and one carry output bit $carry_n$.

B. Answer Set Programming

ASP [17] is a declarative programming framework that is widely used to solve difficult search problems, where the search problems are reduced to computing answer sets.

The basic idea of ASP is to represent a given computational problem by a logic program, whose answer sets corresponds to solutions and use an ASP solver to find the answer sets of a logic program. The logic program is built from basic notions, that correspond to the language of first-order predicate calculus.

We follow standard definitions of propositional ASP [18], [19]. Let l, m, n be a non-negative integers such that $l \leq m \leq n$, $a_1, ..., a_n$ be distinct atoms. We refer by *Literal* to an atom or the negation thereof. Literals are the basic building blocks of logic programs.

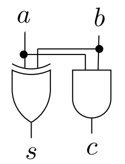
Definition 1 (Logic Program): A *Logic program* Π over a set $\mathcal{A}$ of literals is a finite set of *Rules* in the following form:

$$a_1 \vee ... \vee a_l \leftarrow a_{l+1}, ..., a_m, \neg a_{m+1}, ..., \neg a_n$$

where $l \leq m \leq n$ and each $a_i \in \mathcal{A}$ is a literal, where $1 \leq i \leq n$. For a rule r, we let $H_r := \{a_1, ..., a_l\}$, $B_r^+ := \{a_{l+1}, ..., a_m\}$, $B_r^- := \{a_{m+1}, ..., a_n\}$ and $B_r = B_r^+ \cup B_r^-$. Let r be a rule of Π. Then, r is said to be a *fact*, if and only if $B_r := \emptyset$. We denote the set of *atoms* occurring in a rule r or in a program Π by $at(r) := H_r \cup B_r^+ \cup B_r^-$ and $at(\Pi) := \cup_{r \in \Pi} at(r)$. The answer set (stable model) semantics is defined in terms of a *reduct* of a program Π relative to a set Q of literals.

Definition 2 (Gelfond-Lifschitz Reduct): Let Π be a program and Q be a set of literals. Then, the reduct Π^Q of a program Π is defined as follows:

$$\Pi^Q := \{H(r) \leftarrow B_r^+ \mid r \in \Pi, B_r^- \cap Q = \emptyset\} \tag{3}$$



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

Fig. 1. Half adder logic diagram and its truth table.

To illustrate (3), a reduct Π^Q of a program Π under a set Q of literals is the program obtained by first removing all rules r with $B_r^- \cap Q = \emptyset$ and then removing all $\neg a$ where $a \in B_r^-$ from the remaining rules r. An answer set of a program Π can be defined as follows.

Definition 3 (Answer Set): Let Π be a logic program and Q be a set of literals. Then, Q is an answer set, if and only if $Q = Cn(\Pi^Q)$.

We refer by $AS(\Pi)$ to a set of all answer sets of a program, such that $AS(\Pi) := \{Q \subseteq at(\Pi) \mid Cn(\Pi^Q) = Q\}$. Intuitively, an answer set contains the minimal set of atoms that satisfy the program, and no subset of this set satisfies the program. Therefore, the answer set represents a minimal model of the given problem. It is important to distinguish this approach from classical logic. Unlike in answer set programming, not all models in classical logic are considered to be minimal models.

Example 1: Consider the program of the circuit in Fig. 1:

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

The first four rules captures the xor gate, while the last three rules captures the and gate. As can be seen in Table I, $AS(\Pi) := \{\{a, s\}, \{b, s\}, \{a, b, c\}\}$ w.r.t. a program Π of Example 1, since those are the only sets of literals that satisfy the condition of being an answer set ($Cn(\Pi^Q) = Q$). In Table I, some possible set of literals Q are omitted, as they are not satisfied by the answer set condition.

In the next section, we show how ASP can be used to model a specific representation of an adder circuit. More precisely, we restrict our focus to the *And-Inverter Graph* (AIG) [20] representations that are well-known in synthesis of logic functions.

III. Circuit Modeling Using ASP

The general idea is to represent the behavior of gates and adder functions into ASP rules, and the connections between gates together with the values of inputs as facts. Then, the ASP solver is used to reason about the values of output gates and to verify that each output gate value matches its corresponding

TABLE I
Given a program Π of Example 1, the results of computing a reduct Π^Q of Π under Q, and the smallest set of literals $Cn(\Pi^Q)$ per a set Q of literals.

Q	Π^Q	$Cn(\Pi^Q)$
$\{\}$	$\{s \leftarrow a; s \leftarrow b; a \leftarrow;$ $b \leftarrow; c \leftarrow a, b; \}$	$\{a, b\}$
$\{a\}$	$\{s \leftarrow a; a \leftarrow; c \leftarrow a, b; \}$	$\{a, s\}$
$\{a, s\}$	$\{s \leftarrow a; a \leftarrow; c \leftarrow a, b; \}$	$\{a, s\}$
$\{b, s\}$	$\{s \leftarrow b; b \leftarrow; c \leftarrow a, b; \}$	$\{b, s\}$
$\{a, b, c\}$	$\{c \leftarrow a, b; a \leftarrow; b \leftarrow; \}$	$\{a, b, c\}$
$\{a, b, s, c\}$	$\{c \leftarrow a, b; a \leftarrow; b \leftarrow; \}$	$\{a, b, c\}$

logic function. For the encoding of the circuit as an ASP logic program we rely on the input language of the ASP solver *Clingo* [21], which is an extended version of *Prolog* [22].

To illustrate the encoding of a circuit, the AIG graph representation of a simple adder architecture (e.g., *Ripple Carry Adder* (RCA)) is used. First, it is essential to define the AIG graph G formally. Let and, inv, $input$ and $output$ be a disjoint sets of and, inverter, input and output gates appearing in G, respectively. Also, let $gates$ be a union of all gates. Given a netlist on the reverse topological order (i.e., an output gate is always in a higher order than its inputs), a graph AIG G can be seen as a *Directed Acyclic Graph* (DAG), which is defined as follows.

Definition 4 (AIG Graph): Let $G = (V, E)$ be a directed acyclic graph such that:

- $V := \{v \mid v \in gates\}$.
- $E := \{(v, v') \mid v, v' \in V, v' \text{ is reachable from } v\}$.

In order to model the circuit, the behavior of gates is modeled using ASP rules, such that gate behavior is defined based on values on their ports. These ports provide a mechanism to handle 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. A binary predicate symbol $conn(P1, P2)$ is used to define connection between ports $P1$ and $P2$.

Since an AIG graph has different types of gates, we use a binary predicate symbol $type(G, t)$ to label a gate G with a type t. e.g., *and*. Fig. 2a illustrates an AIG for a ripple carry adder with 2-bit inputs.

To convert facts and rules that are introduced in Eq. (3) into the clingo language, we use the following mapping rules:

- A fact $p \leftarrow$ is mapped to p.
- A rule $a \leftarrow b_1, ..., b_n$ is mapped to $a : -b_1, ..., b_n$.

Clingo also provides an interface to represent logical operations. *and*, *or*, and *xor* logic functions are represented by symbols "&", "?" and "ˆ", respectively. Due to the restrictions of the AIG representation, it is required to represent only *and* and *inverter* gates. The *and* gate can be characterized as follows:

$$\begin{aligned} val(out(G), X\&Y) : -type(G, and),& \\ val(in1(G), X), val(in2(G), Y).& \qquad (4) \end{aligned}$$

As the AIG graph is restricted to *and* gates with two inputs and one output, the unary functions $out(G), in1(G)$, and $in2(G)$ are used to represent the output, first and second input ports, respectively. However, we only need one output and one input to characterize *inverter* gate behavior. The unary functions $out(G)$ and $in(G)$ are used to handle the output and input ports, respectively. The *inverter* gate can be characterized as follows:

$$val(out(G), 1\hat{\ }X) : -type(G, inverter), val(in(G), X). \qquad (5)$$

In Eq. (5), the *xor* logical operation is used to represent the negation of the value of X. Finally, the connection between two ports is defined as follows:

$$val(P2, V) : -conn(P1, P2), val(P1, V). \qquad (6)$$

The intuitive meaning of the previous rule is that if port $P1$ is connected to $P2$ and $P1$ has value V, then $P2$ has value V. It is worth noting that the value of a port is restricted to 0 and 1. Those values appear as constants in the program and are passed from one of primary inputs that is connected to a gate port. Therefore, we enable representing primary inputs as gates with only one port, where they are characterized by facts indicating their value. E.g., facts $val(a_0, 0)$ and $val(b_0, 1)$ indicate that gates a_0 and b_0 have values 0 and 1, respectively. The primary outputs are represented analogously, except that their values are observed from the circuit. Informally, by Eq. (6), values of ports are passed from the primary inputs to other gates until they reach the primary outputs. Hence, to ensure the correctness of an adder circuit, it is essential to check, whether the value of each output gate satisfies the adder function as shown in Eq. (1) and Eq. (2). Therefore, it is essential to encode sum and carry functions into clingo. They can be characterized as follows:

$$\begin{aligned} sum(sum_i, V) : -val(a_i, A), val(b_i, B),& \\ carry(carry_{i-1}, C), V = A\hat{\ }B\hat{\ }C.& \qquad (7) \\ carry(carry_i, V) : -val(a_i, A), val(b_i, B),& \\ carry(carry_{i-1}, C), V = (A\&B)?(C\&(A\hat{\ }B)).& \qquad (8) \end{aligned}$$

Equations (4), (5) and (6) are very general and can work independently of the circuit architecture, while Eq. (7) and Eq. (8) are only related to adder circuits. However, in order to complete the model, we further have to add facts representing the structure of the circuit. E.g., $conn(out(and4), in1(and16))$ represents the connection between the output port of gate "4" and the first input port of gate "16". It is worth noting that those facts are circuit dependent. E.g., $carry(carry_{-1}, 0)$, $type(and2, and)$, $type(inv1, inverter)$ and $conn(out(inv1), in1(and2))$.

Finally, to enable the verification of output gates, it is necessary to relate output gates with their expected logic functions representing adder functions (see Eq. (7) and Eq. (8)). Thus, we introduce one clingo rule per output bit to reach the desired behavior. Considering Fig. 2a, the clingo rules for the verification of all outputs can be summarized as follows:

$$\begin{aligned} \textit{verify}(o_0) : -sum(sum_0, X), val(o_0, X).& \\ \textit{verify}(o_1) : -sum(sum_1, X), val(o_1, X).& \\ \textit{verify}(o_2) : -carry(carry_1, X), val(o_2, X).& \qquad (9) \end{aligned}$$

The idea behind Eq. (9) is that for a given set of facts representing an input sequence of the primary inputs, output bit i is said to be correct, if $\textit{verify}(o_i)$ appears in the answer set of the program. This can be formulated as follows.

Definition 5 (Valid Sequence): Let S be a set of facts representing an input sequence of n inputs. Then, S is said to be a *valid sequence*, if and only if there exists an answer set $Q \in AS(\Pi)$ such that $\bigcup_{i=0}^{n}\{\textit{verify}(o_i)\} \cup S \subseteq Q$.

By Definition 5, if the input sequence is correct, their verification atoms $\textit{verify}(o_i)$ will appear in one of the answer sets of the program Π. Due to the fact that all input sequences must be a valid sequence to be able to ensure correctness of a circuit. The previous definition can be generalized as follows.

Definition 6 (Valid Graph): Let Π be a program defined w.r.t. AIG graph G of size n, $\mathcal{F}$ be a set of sets of facts such that each $s \in \mathcal{F}$ represents an input sequence, and $|s| = n$. Then, G is said to be a *valid graph*, if and only if for every $s \in \mathcal{F}$, there exists an answer set Q such that s is a valid sequence. Otherwise, G is an *invalid graph*.
It is worth noting that the search space is 2^n, and consequently $|\mathcal{F}| = 2^n$. Also, $|s| = n$, for all $s \in \mathcal{F}$.

In the next section, we propose an approach for achieving formal verification of a circuit in linear time, by applying dynamic programming on graph G to obtain an upper bound of the search space.

IV. Polynomial Formal Verification of Adder Circuits

In this section, we introduce an approach for splitting an AIG graph G into subgraphs, which relies on the idea from [13] and we propose a method for subgraph reduction. Subsequently, we present an approach for passing information between the subgraphs. Based on that, we define the verification of the subgraphs. We assume familiarity with graphs and trees [23].

A. Graph Unraveling and Reduction

Given an AIG graph $G = (V, E)$ and a node $v \in V$, a subgraph (G, v) can be constructed as follows.

Definition 7 (Subgraph): Let $G = (V, E)$ be a graph, $v \in V$ be a node. Then, a subgraph $(G, v) = (V_v, E_v)$ of G is obtained such that:

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

In graph theory, the cutwidth of a graph $G = (V, E)$ w.r.t. a nodes ordering h is the smallest integer k such that for every $l = 1, ..., |V| - 1$, there exist at most k edges with one endpoint in $\{v_1, ..., v_l\}$ and the other endpoint in $\{v_{l+1}, ..., v_{|V|}\}$, where $\{v_1, ..., v_{|V|}\} \in V$. In other words, the cutwidth for some vertices ordering is the size of the largest cut induced by that ordering.

We refer to the nodes induced by the cut as *cone nodes*. We use the notion "node" to refer to a gate of the circuit. AIG graph G can be seen as a multi-root tree such that each root node represents an output bit. Therefore, it is possible to split G of size n into n subgraphs by taking one node $v \in V$ representing an output bit and traversing all nodes v', that are reachable from v as shown in Fig. 2. It is worth noting that there exist nodes that appear in several subgraphs. e.g., node "4" appears in graphs (G, O_0), (G, O_1) and (G, O_2). We define the set of cone nodes appearing in a sub-graph as follows.

Definition 8 (Cone Nodes): Let $(G, v) = (V_v, E_v)$ be a subgraph of $G = (V, E)$. A set $C_{(G,v)}$ of cone nodes defined w.r.t. (G, v) such that $C(G, v) := \{a \in V_v \mid (b, a) \in E, b \in V \setminus V_v\}$. Let $C_i := \bigcup_{j=0}^{i} C(G, v_i)$, and $C(G) := \bigcup_{i=0}^{n} C(G, v_i)$, where n is the number of output nodes.

To check whether each subgraph is valid, the definition of a valid graph from Definition 6 is used. However, the values of cone nodes are evaluated multiple times. e.g., node "4" is computed in all subgraphs.

To be able to bound the number of inputs of each sub-graph and overcome the problem of evaluating the cone node more than once, we propose a reduction of the sub-graph based on $C(G)$ to obtain such a bound.

Definition 9 (Reduced Subgraph): Let $(G, v_i) = (V_{v_i}, E_{v_i})$ w.r.t. node v_i representing output gate i, where $0 < i \leq n$. A reduced subgraph $R(G, v_i) = (R(V_{v_i}), R(E_{v_i}))$ is a sub-graph of (G, v_i) such that:

- $R(V_{v_i}) := \{a \in V_{v_i} \mid a \notin C_{i-1}\}$.
- $R(E_{v_i}) := \{(a, b) \mid a, b \in R(V_{v_i}), b \text{ is reachable from } a\}$.

By Definition 9, the nodes of the resulting subgraphs are disjoint. To adapt the notion of the input node with the reduced subgraph, we refer to any node of the reduced subgraph with no successor as an input node. For simplicity, we use R_{G_i} to refer to the reduced subgraph $R(G, v_i)$, where i is referring to output bit i. We further denote the inputs of R_{G_i} as IN_i, which is split into the primary inputs PIN_i and the incoming cone nodes from other subgraphs CIN_i. The primary output is referred to as OUT_i and the outgoing cone nodes as $COUT_i$.

We adapt the notion of a *k-bounded circuit* introduced in [24] to the case of graphs. Briefly, a *circuit* is said to be *k-bounded* if its nodes can be partitioned into disjoint blocks such that each block has at most *k* inputs. Thus, Definition 9 yields a characterization of *k-bounded circuit* in terms of the graph.

Definition 10 (k-bounded Graph): Let $G = (V, E)$ be a graph of n root nodes. Then, G is said to be *k-bounded*, if and only if for all $0 \leq i \leq n$, $R(G, v_i)$ has at most k input nodes.

B. Information Passing

As we can see in Fig. 2, each cone node v is evaluated in one of the subgraphs only. Also, any other graph that uses v as an input, takes the value of v from the graph in which it is evaluated. For example the node "4" is calculated in subgraph (G, O_0) but is also used in the reduced subgraph (G, O_1). Hence, the cone node values of reduced subgraphs must be stored, so their values can be used in other subgraphs. The value cannot be stored as a function over the primary inputs, because this would pass the primary inputs from one subgraph to the next and the last subgraph would be dependent on all primary inputs. Following the example, if node "4" would be stored as $B_0 \wedge A_0$, the reduced subgraph (G, O_1) and its outgoing cone nodes would be dependent on A_0, A_1, B_0, B_1 and the list would grow by two elements for every subgraph. To overcome this issue we use the carry function to store information about the cone nodes. A hash table relates the values of the cone nodes with the corresponding value of the carry function. This allows us to use the carry function in the specification of the output for the subgraph. To define this table, we first introduce an injective function f, which maps the input sequence $s \in IN_i$ of subgraph R_{G_i} to the set of values $COUT_i$ of outgoing cone nodes $C(G_i)$.

$$f \colon IN_i \mapsto COUT_i. \tag{10}$$

The surjective function g that maps $c \in COUT_i$ to the value of the *carry* function.

$$g \colon COUT_i \mapsto [0, 1]. \tag{11}$$

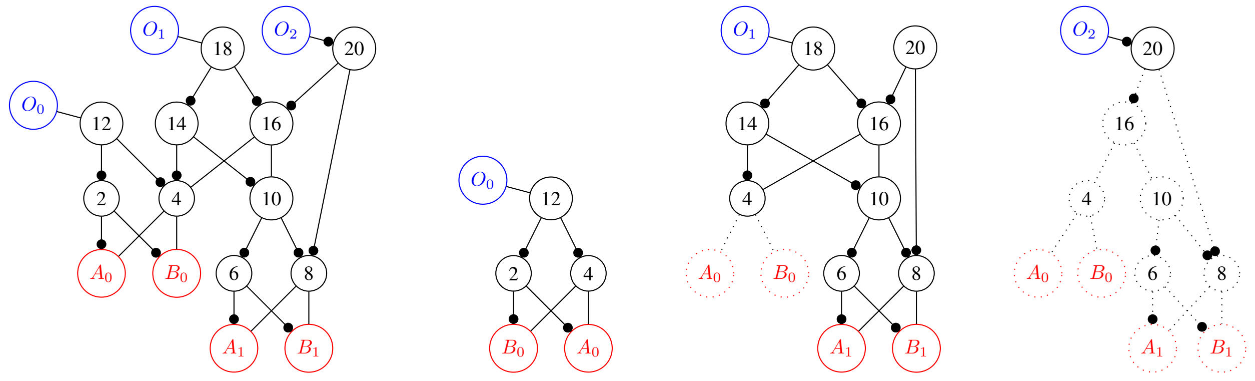


Fig. 2. AIG of 2-bit RCA and the resulting (reduced) subgraphs (G, O_0), (G, O_1) and (G, O_2) that can be obtained from the AIG graph in Fig. 2a. The nodes highlighted in red correspond to input nodes, and those highlighted in blue correspond to output nodes. Dotted nodes and edges are removed when the subgraph is reduced

TABLE II
$\mathcal{T}_0$ FOR FIG. 2

$COUT_0$	Value
{0}	0
{1}	1

We refer by $f(s)$ to the set of values representing cone nodes $C(G_i)$ under the input sequence $s \in IN_i$. Also, by $g(f(s))$ to the value of $carry$ under $f(s)$. The key of a hash table entry is the value $f(s)$ for which there is a value $s \in IN_i$, while the corresponding value in the hash table is $g(c)$. Equation 12 defines the hash table accordingly.

$$\mathcal{T}_i = \{(f(s), g(f(s))) \mid s \in IN_i\}. \tag{12}$$

Let us examplarily create the hash table $\mathcal{T}_0$ for Fig. 2. There are two possible values of f for input sequences $s \in COUT_0$: $f(0, 1) = \{0\}$ and $f(1, 1) = \{1\}$. We do not have to consider the other input sequences as both possible combinations for the set of $COUT_0$ are already covered, because the set of cone nodes only contains node "4". The carry function for subgraph 0 is $A_0 \wedge B_0$. For function g we can obtain that $g(\{1\}) = 1$ and $g(\{0\}) = 0$. The resulting hash table is presented in Table II. Of course the hash tables get more complex, if the set of cone nodes is bigger. To generalize the information passing from adder functions to arbitrary function, the carry function can be swapped with one or multiple functions, which describe the information passed between the subgraphs. This does not increase the number of entries in the hash table, as they are bound by the number of cone nodes.

C. Subgraph Verification

For every subgraph R_{G_i} two tasks have to be performed. First it has to be verified that the output function of the subgraph is correct and second the hash table $\mathcal{T}_i$ for $COUT_i$ of the circuit has to be built.

The input IN_i contains the primary inputs PIN_i and the cone nodes CIN_i. It is important to only allow combinations from CIN_i which are in $\mathcal{T}_j$. The values of CIN_i of R_{G_i} may be stored in any hash table T, where $j < i$. Thus, it is required to go over all tables j to obtain such values. Therefore, a relation has to be defined between two tables $\mathcal{T}_j$ and $\mathcal{T}_{j'}$ of subgraphs, where $j, j' < i$. A relation $\bowtie$ is used to define the relation w.r.t. CIN_i between two tables $\mathcal{T}_j$ and $\mathcal{T}_{j'}$ such that $\mathcal{T}_j \bowtie \mathcal{T}_{j'} := \{r \cup r' \mid r \in \mathcal{T}_j, r' \in \mathcal{T}_{j'}, C(G_j) \cap C(G_{j'}) \subseteq CIN_i\}$. Hence, we refer by $\mathcal{X}_i(CIN_i)$ to the resulting table containing the values of CIN_i and defined as follows.

$$\mathcal{X}_i(CIN_i) := \mathcal{T}_{i-1} \bowtie \ldots \bowtie \mathcal{T}_0. \tag{13}$$

Finally, every $r \in \mathcal{X}_i(CIN_i)$ is populated with PIN_i of R_{G_i} to obtain its input sequences.

Thus, each reduced subgraph R_{G_i} can be checked independently whether it is a valid graph (recall Definition 6), for all $0 \leq i \leq n$, where n is the number of outputs. Moreover, the outgoing hash table $\mathcal{T}$ of each subgraph can be built. In the following section, we show the overall time complexity of the proposed approach.

V. Time Complexity

We refer by $\Pi(R_{G_i})$ to a logic program constructed w.r.t. a reduced subgraph R_{G_i}, and by $\Pi(G)$ to a logic program constructed w.r.t. the input AIG graph G. Then, checking the graph validity of $\Pi(R_{G_i})$ depends on the number of its input nodes IN_i. Thus, we obtain the following theorem.

Theorem 5.1: Let R_G be a reduced subgraph. Then, $\Pi(R_G)$ can be verified in time $\mathcal{O}(2^{|IN|})$.

Proof: By Definition 6, R_G is a valid graph if and only if for every $s \in \mathcal{F}$, we have that s is a valid sequence, where $\mathcal{F}$ is the set of all input sequences. Also, the size of $\mathcal{F}$ depends on the number of input nodes IN of R_G and the carry of its previous subgraph (for the reduced subgraph R_{G_i}, where $i > 0$). Therefore, the overall number of input sequences is $2^{|IN|}$ and consequently, $\Pi(R_G)$ has search space of $2^{|IN|}$. Hence, $\Pi(R_G)$ can be verified in time $\mathcal{O}(2^{|IN|})$. ■

Since each reduced subgraph R_{G_i} could contain a node c such that c is a cone node and the values of c are stored in $\mathcal{T}_j$ where $j < i$, the values of c can be obtained from $\mathcal{X}_i(CIN_i)$ (recall Eq. (13)). We assume that $\mathcal{X}_i(CIN_i)$ can be computed in constant time. This assumption is done based on the fact that the search operation in a well configured hash table takes

TABLE III

CALCULATED UPPER BOUND AND THE MAXIMUM NUMBER OF *and* GATES AMONG ALL REDUCED SUBGRAPHS FOR DIFFERENT ADDER CIRCUITS.

Adder	**Upper bound (K)**	**Maximum No. *and* Gates**
RCA	3	7
CSKA	8	15
CLA	11	18
CSA	N.A.	N.A.
KSA	N.A.	N.A.
LFA	N.A.	N.A.

constant time. In the worst case the operation can take linear time, but only if many collisions occur i.e. as a result of a bad hash function. Consequently, $\mathcal{X}_i(CIN_i)$ can be computed in constant time (denoted by $P(\mathcal{X}_i(CIN_i))$).

Finally, the overall time required to verify $\Pi(G)$ of the AIG graph G can be characterized in the following theorem.

Theorem 5.2: Let G be an AIG graph constructed w.r.t. an adder circuit. Then, $\Pi(G)$ can be verified in time $\mathcal{O}(n \cdot 2^K)$, where n is the input bit width and K is the maximum size of input nodes of all reduced subgraphs.

Proof: Let G be a graph of n input bit width, then n subgraphs (G, v_i) have to be constructed from G by Definition 7, where $0 \leq i \leq n$. Also, the reduced subgraph R_{G_i} can be obtained from (G, v_i) by applying Definition 9. By Definition 9, R_{G_0} is equivalent to (G, v_0) ($V_{v_0} = R(V_{v_0})$ and $E_{v_0} = R(E_{v_0})$). Thus, the set $CIN_0 = \emptyset$. Therefore, R_{G_0} relies only on the primary input nodes PIN_0. More precisely, by Theorem 5.1, $\Pi(R_{G_0})$ can be verified in time $\mathcal{O}(2^{|PIN_0|})$. However, in order to enable verifying the reduced subgraph R_{G_i}, it is essential to compute $\mathcal{X}_i(CIN_i)$ (Equation 13) where $0 < i \leq n$. This is due to the fact that for all R_{G_i}, where $i > 0$, and $CIN_i \neq \emptyset$. Since $\mathcal{X}_i(CIN_i)$ is computed from tables $\mathcal{T}_j$, where $j < i$. Let P_j be the constant time required for a single access of table $\mathcal{T}_j$. Then the overall time complexity for computing $\mathcal{X}_i(CIN_i)$ can be calculated as follows:

$$\mathit{Complexity}(\mathcal{X}_i(CIN_i)) := \sum_{j=0}^{i-1} P_j \tag{14}$$

We refer by $P(\mathcal{X}_i(CIN_i))$ to the time obtained from the previous equation. By Theorem 5.1, $\Pi(R_{G_i})$ can be verified in time $\mathcal{O}(2^{|IN_i|})$, where $|IN_i| := |CIN_i| + |PIN_i|$. Thus, the overall verification process of subgraph R_{G_i} has a time complexity of $\mathcal{O}(2^{|IN_i|} + P(\mathcal{X}_i(CIN_i))) = \mathcal{O}(2^{|IN_i|})$, where $0 < i \leq n$. The overall time complexity for verifying $\Pi(G)$ can be calculated as follows:

$$\mathit{Complexity}(\Pi(G)) := \sum_{i=0}^{n} \mathcal{O}(2^{|IN_i|}) \tag{15}$$

Moreover, by Definition 10, G is said to be *k-bounded* if and only if every reduced subgraph R_{G_i} has at most k-input nodes. Let K be the maximum size of input nodes of all reduced subgraphs. Equation 15 shows that $\Pi(G)$ can be verified in time $\mathcal{O}(n \cdot 2^K)$. Hence, if K is constant, then the graph G can be verified in a linear time. ■

VI. EXPERIMENTAL WORK

To evaluate the upper bound K for the verification process of adder circuits introduced in Section V and check the scalability of our approach, we have implemented the ASP framework in Python. It is worth noting that our approach is not restricted to a specific architecture. The framework takes input circuit in the standard AIGER format [25]. The verification is performed independently for each subgraph. This allows us to detect an incorrect circuit without necessitating a complete verification of the entire circuit.

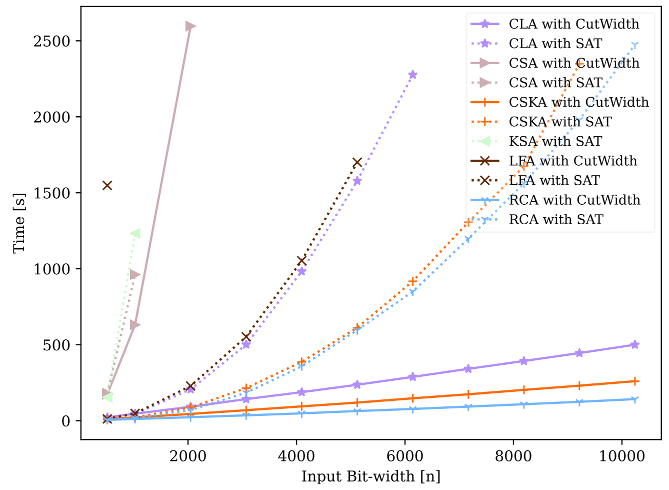


Fig. 3. Runtime graphs per adder circuit. The x-axis refers to the input bit-width, and the y-axis depicts the runtime sorted in ascending order for each circuit type individually. The solid lines indicate the runtime obtained from CutWidth approach, while the dotted lines indicate the one obtained from SAT approach.

A. Experimental Setup

We mainly compare our approach (labeled as *CutWidth*) and SAT-based approach (labeled as *SAT*) in terms of the wall clock time and the number of timeouts. All instances are performed on Intel(R) Core(TM) i7-11370 with 3.30 GHz. We set a timeout of 2700 seconds and a limited available RAM to 16 GB per instance. We use different types of bug-free adder circuits of different sizes (RCA, CSKA, CLA, CSA, KSA, and LFA). These circuits are generated using the ArithsGen tool [26], where the design is synthesized using yosys.

B. Experimental Results

Table III shows the results of the upper bound of inputs (second column) and the maximum number of and gates appearing in reduced subgraphs (third column) for adder architectures (first column) during the verification process. The value *N.A.* indicates that the circuit does not have a constant cutwidth. We can observe that the upper bound depends on the circuit architecture. E.g. *RCA* has three input nodes, representing two primary inputs and the previous carry. This is due to the fact that each circuit has a different architecture, and consequently the number of cone nodes that appear as inputs of reduced subgraph is circuit dependent. A circuit with a constant cutwidth has a fixed number of *and* gates.

Table IV compares the run time of each approach for each adder architecture w.r.t. different input size, where the input size is represented in the first column, while the other columns indicate the runtime of the adder architectures under SAT and CutWidth approaches. If an approach did not terminate within the timeout limit, the runtime of this instance is set to *T.O.*. It

TABLE IV
RUN TIME OF VERIFYING ADDER CIRCUIT (SECONDS).

Size	Benchmarks											
	RCA		CSKA		CLA		CSA		KSA		LFA	
	SAT	CutWidth	SAT	CutWidth	SAT	CutWidth	SAT	CutWidth	SAT	CutWidth	SAT	CutWidth
512	5.5	5.9	7.1	11.2	13.0	22.6	180.2	174.6	153.9	T.O.	12.2	1548.4
1024	17.6	11.5	22.1	22.1	44.9	46.3	962.1	631.4	1231.8	T.O.	48.1	T.O.
2048	71.0	23.3	87.8	45.0	206.7	92.1	T.O.	2596.3	T.O.	T.O.	228.7	T.O.
3072	184.0	35.5	214.9	69.4	501.3	142.3	T.O.	T.O.	T.O.	T.O.	552.5	T.O.
4096	357.9	48.7	387.3	94.5	982.7	188.1	T.O.	T.O.	T.O.	T.O.	1053.2	T.O.
5120	596.4	63.9	612.3	119.6	1578.8	236.9	T.O.	T.O.	T.O.	T.O.	1700.9	T.O.
6144	850.3	77.5	917.9	147.8	2276.6	288.6	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
7168	1198.1	93.0	1306.2	174.1	T.O.	341.5	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
8192	1561.6	108.7	1673.8	203.1	T.O.	393.1	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
9216	1979.8	124.6	2348.8	230.8	T.O.	445.5	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
10240	2475.1	142.3	T.O.	260.3	T.O.	500.3	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.

shows that the CutWidth approach is faster for adder circuits with a constant cutwidth than the SAT approach, while the SAT approach outperforms the CutWidth one for the circuits that have no constant cutwidth. E.g., the SAT approach was able to solve instances up to 5k inputs, while the CutWidth approach reached the timeout, starting from 1k inputs.

Moreover, Fig. 3 shows the run time of each approach for each adder architecture per input size, where the run time per input size is shown in Table IV. Hence, the curve of the CutWidth approach for adders with a constant cutwidth is a linear curve. Therefore, it aligns with the calculated complexity bound obtained from Theorem 5.2 in Section V. Also, the curve of SAT approach has an exponential behavior.

VII. CONCLUSION

In this paper, we have proposed a new PFV approach that relies on the cutwidth of a netlist, where ASP was used to verify subcircuits independently and reason about nodes that are used in more than one subcircuit. Moreover, we have shown that the verification of adder circuits can be done in linear time, for a constant cutwidth K. Finally, the experimental evaluations confirm the upper bound complexity of each circuit.

As future work, we will focus on extending this approach for the PFV of combinational adder circuits to the case of sequential adder circuits. Moreover, our study aims to examine various circuit types to determine, which ones exhibit a constant cutwidth. Furthermore, we plan to apply different verification techniques for the verification of the subcircuits. While SAT solvers can be expected to behave similarly to ASP solvers, using BDD is particularly interesting. The main challenge will be adapting the hash table $\mathcal{T}$.

REFERENCES

[1] A. Gupta, M. K. Ganai, and C. Wang, "SAT-Based verification methods and applications in hardware verification," in *Formal Methods for Hardware Verification*, 2006, pp. 108–143.

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

[3] J. Harrison, "Theorem proving for verification (invited tutorial)," in *CAV*, 2008, pp. 11–18.

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

[5] G. Brewka, T. Eiter, and M. Truszczyński, "Answer set programming at a glance," *ACM*, vol. 54, no. 12, p. 92–103, 2011.

[6] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Answer Set Solving in Practice*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning, 2012.

[7] A. Provetti and T. C. Son, "Answer set programming: Towards efficient and scalable knowledge representation and reasoning," in *Proceedings of the 1st Intl. ASP'01 Workshop*, 2001.

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

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

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

[11] D. M. Thilikos, M. Serna, and H. L. Bodlaender, "Cutwidth i: A linear time fixed parameter algorithm," *Journal of Algorithms*, vol. 56, no. 1, pp. 1–24, 2005.

[12] A. C. Giannopoulou, M. Pilipczuk, J. Raymond, D. M. Thilikos, and M. Wrochna, "Cutwidth: Obstructions and algorithmic aspects," *Algorithmica*, vol. 81, no. 2, pp. 557–588, 2019.

[13] M. R. Prasad, P. Chong, and K. Keutzer, "Why is combinational ATPG efficiently solvable for practical VLSI circuits?" *JETTA*, vol. 17, pp. 509–527, 2001.

[14] A. Mahzoon and R. Drechsler, "Late breaking results: Polynomial formal verification of fast adders," in *DAC*, 2021, pp. 1376–1377.

[15] ——, "Polynomial formal verification of prefix adders," in *ATS*, 2021, pp. 85–90.

[16] C. Wolf, "Yosys open synthesis suit," available at https://github.com/YosysHQ/yosys, 2022.

[17] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," *Logic Programming*, vol. 2, 2000.

[18] V. W. Marek and M. Truszczynski, "Stable models and an alternative logic programming paradigm," *A Computing Research Repository*, 1998.

[19] I. Niemelä, "Logic programs with stable model semantics as a constraint programming paradigm," *Annals of Mathematics and Artificial Intelligence*, vol. 25, pp. 241–273, 1999.

[20] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *DAC*, 2006, pp. 532–535.

[21] M. Gebser, R. Kaminski, A. König, and T. Schaub, "Advances in gringo series 3," in *LPNMR*, 2011, pp. 345–351.

[22] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, 1987.

[23] R. Diestel, *Graph Theory*, 4th ed. Springer, 2010, vol. 173.

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

[25] A. Biere, "The AIGER And-Inverter Graph (AIG) format version 20071012," Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep. 07/1, 2007.

[26] J. Klhufek and V. Mrazek, "Arithsgen: Arithmetic circuit generator for hardware accelerators," in *DDECS*, 2022, pp. 44–47.