Polynomial Formal Verification of Multi-Valued Logic Circuits within Constant Cutwidth Architectures

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

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

***Abstract*—Formal verification is essential for circuit correctness. Extending binary logic verification to *Multi-Valued Logic* (MVL) presents challenges due to encoding challenges. *Answer Set Programming* (ASP) enables compact MVL circuit encoding. In this paper, we propose a *Polynomial Formal Verification* (PFV) approach for MVL circuits with a constant cutwidth. It employs a divide-and-conquer algorithm to decompose circuits into subcircuits, each representing an output, where ASP is used to encode and verify the subcircuits. The approach reduces verification complexity to the circuit's cutwidth, independent of input bitwidth, ensuring linear time verification for circuits with constant cutwidth. We validate our approach using adder architectures, as many adder architectures have a constant cutwidth. Empirical experiments involve various adder architectures and different logic levels.**

***Index Terms*—Polynomial Formal Verification, Logic Synthesis, Multi-Valued Logic, Answer Set Programming, Cutwidth.**

I. Introduction

In the realm of computer system design, ensuring the correctness of circuit behavior is a challenging problem. This problem arises from the increasing complexity of circuit designs. Consequently, various formal verification techniques [1], [2] have been introduced to validate the functional behavior of circuits.

These techniques include efficient logic function representations like *Binary Decision Diagrams* (BDDs) [3], [4] or *Multi-valued Decision Diagrams* (MDDs) [5], or encoding of the verification problem as an SAT instance using miter circuits [6]. However, these techniques challenge a computational complexity problem [7]. Therefore, *Polynomial Formal Verification* (PFV) [8] has been introduced to provide an upper bound of the time complexity. It has shown earlier in [9]–[11] for the binary logic that several types of circuits can be verified in polynomial time.

In this paper, we focus on the circuits that exhibit a specific structural property. More precisely, we consider the circuits of a limited *Cutwidth* [12], [13]. In the area of formal verification, cutwidth corresponds to the minimum number of edge-cuts required to split the circuit into subcircuits.

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

The paper introduces a novel PFV approach to verify the specific circuits within constant cutwidth architectures. The approach includes dividing the circuit into subcircuits and storing the interleaving information. This enables the independent verification of each subcircuit. Moreover, these subcircuits are modeled and verified into ASP [14], [15]. Unlike SAT, ASP allows to provide a compact representation of MVL functions. Specifically, ASP allows for the direct encoding of non-binary values of gates directly within the modeling language.

In prior work [16], it has been proven that the verification of circuits with a constant cutwidth in binary logic can be done in linear time. We show that these results can be generalized to the MVL domain.

Due to the fact that several adder architectures exhibit a constant cutwidth, we conduct experimental work for adders to demonstrate our theoretical findings. It has been proven earlier in [17] that the verification process of adder circuits in MVL using MDDs is possible in polynomial time, but linear time could not be proven. However, this work lacked empirical validation. Our approach for adder circuits represents both a theoretical improvement for the state-of-the-art and empirical validation. While our approach contributes to the theoretical aspect, we also conduct experiments to confirm our theoretical findings. Our experiments demonstrate the performance for various types of adder architectures, involving up to 10k input bits and different logic levels up to four-valued logic.

The paper is structured as follows: In Section II we introduce MVL operators, cutwidth as a structural property of the *And-Inverter Graph* (AIG) [18] representation of a circuit, MVL addition w.r.t. MVL operators and AIG, and the basic concepts of ASP. Subsequently, the modeling of circuits in ASP is described in Section III. Section IV extends the approach [16] to the multi-valued domain. Section V describes the complexity properties of our approach. This is followed by an experimental evaluation in Section VI.

II. Preliminaries

A. Multi-Valued Operators

In this section, we define the basic multi-valued operators for representing and manipulating MVL functions.

Definition 1 (MVL Operators): Let $a, b, x \in \{0, 1, ..., p-1\}$ be integers, where p is the logic level. Then, the operators $\cdot, +, \neg, \oplus$ are defined as follows [19]:

- $a \cdot b = min(a, b)$.
- $a + b = max(a, b)$.
- $\neg x = (p-1) - x$.
- $a \oplus b = (a \cdot \neg b) + (\neg a \cdot b)$.

It is worth noting that these operators can be used for the binary case (i.e., $p = 2$), where operators $\cdot, +, \neg$, and $\oplus$ correspond to the logical *AND*, *OR*, *Inverter* (denoted by *Inv*), and *XOR*, respectively.

B. Cutwidth of AIG

A circuit C can be seen as a directed acyclic graph *AIG* G, consisting of the sets of inputs PI, outputs PO, *AND* gates, and *Inv* gates. This is defined as follows:

Definition 2 (AIG Graph): Let $G = (V, E)$ be a *AIG* of a circuit C such that:

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

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 $v_{i+1}, ..., v_n$. The set of nodes induced by the *Cut* is referred as *Cone Nodes* (also called *Out-going Nodes*). This leads to a characterization of *K-bounded Graph* that is defined in [20] as follows:

Definition 3 (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, we have the number of inputs of G_i is at most K.

C. Multi-Valued Addition

In this section, MVL operators and the previously defined AIG graph of the adder circuit are used to define the MVL addition function.

Let $a, b, c_{-1} \in \{0, 1, ..., p-1\}$ such that a, b are two inputs with size n bits, and c_{-1} is the incoming carry bit. The addition function Add_i adds two inputs a_i and b_i together with the incoming carry bit c_{i-1}, and outputs the sum s_i and the carry c_i, for all $0 \leq i \leq n$. The sum and carry functions can be characterized as follows:

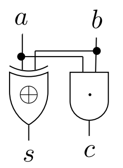
$$s_i = a_i \oplus b_i \oplus c_{i-1} \tag{1}$$

$$c_i = (a_i \cdot b_i) + (c_{i-1} \cdot (a_i \oplus b_i)) \tag{2}$$

As the addition function adds two n-bit numbers with the carry c_{-1}, it results n bits that represent the sum s_n and one carry bit c_n. Therefore, the addition function has $2n+1$ input bits and $n+1$ output bits.

D. Answer Set Programming

ASP is a well-known declarative programming framework from the area of knowledge representation and non-monotonic reasoning [21]. It is mainly used to solve NP-hard search problems while allowing a compact modeling [22], and the search problems are reduced for computing *Answer Sets*. We follow the standard definitions of propositional ASP [23].



a	b	s	c
0	0	0	0
0	1	1	0
0	2	2	0
⋮	⋮	⋮	⋮
2	2	0	2

Fig. 1. Half adder logic diagram and its truth table, where $p = 3$.

We consider a set U of propositional *Atoms*. A *Literal* is an atom $a \in U$ or its negation $\neg a$. A (logic) program is defined in terms of rules over the set U as follows:

Definition 4 (Logic program): Let U be the set of atoms, and l, m, n be non-negative integers such that $l \leq m \leq n$. A *Logic Program* Π is a set of *Rules* of the form $a_1 \vee ... \vee a_l \leftarrow a_{l+1}, ..., a_m, \neg a_{m+1}, ..., \neg a_n$, where $a_1, ..., a_n \in U$.

We refer by $H_r := \{a_1, ..., a_l\}$, $B_r^+ = \{a_{l+1}, ..., a_m\}$, and $B_r^- = \{a_{m+1}, ..., a_n\}$ to the head of r, the positive body of r, and the negative body of r, respectively. Also, we denote the sets of atoms occurring in a rule r by $at(r) := H_r \cup B_r^+ \cup B_r^-$, while the set of atoms occurring in Π is denoted by $at(\Pi) := \cup_{r \in \Pi} at(r)$. A rule r is said to be a *Fact* (*Negation-free*) if $B_r^- = \emptyset$. A set A of atoms satisfies a rule r, if $(H_r \cup B_r^-) \cap A \neq \emptyset$ or $B_r^+ \setminus A \neq \emptyset$. Consequently, the set A is a *Model* of Π, if A satisfies all rules $r \in \Pi$. The *Gelfond-Lifschitz* (GL) *Reduct* [24] of a program Π under a set A of atoms (denoted by Π^A) is obtained by first removing all rules r with $B_r^- \cap A \neq \emptyset$, and then removing all $\neg a$ from the remaining rules r, where $\neg a \in r$. The answer set of a program Π is defined as follows:

Definition 5 (Answer Set): Given a set A of atoms, and a program Π, then A is an answer set of Π iff A is a minimal model Π^A.

We refer by $AS(\Pi)$ to the set of all answer sets of Π. To illustrate this, consider the example of a half adder in Fig. 1. For simplicity, the program Π is constructed for the binary values of the truth table follows.

Example 1: Consider the program Π:

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

The set $AS(\Pi) := \{\{a, s\}, \{b, s\}, \{a, b, c\}\}$ w.r.t. Π, where the answer sets correspond to the 1 values of the truth table.

In the following section, we illustrate the modeling of the circuit into ASP, as well as the MVL values of gates. More precisely, we focus on modeling the AIG representation of the circuit, where the MVL operators introduced earlier (recall Definition 1) are used to enable manipulating the non-binary values over gates.

III. Circuit Modeling Using ASP

The general idea is to encode the behavior of the MVL gates and addition function into ASP rules, while the connections and the MVL values of the inputs as facts. Then, the ASP solver is used to reason about the values of output gates and verify the graph by checking whether each output gate matches its corresponding MVL addition function. We follow the standard modeling language of a well-known ASP solver *Clingo* [25] for modeling and verification of the AIG graph. We strict our focus on the AIG of 2-bit *Ripple Carry Adder* (RCA) that can be seen in Fig. 2(b).

The AIG graph is modeled such that a gate behavior is defined based on the values of their ports. These ports allow for handling the passing of the values between gates. Thus, let $P(G)$ be a unary function symbol representing a port of the gate G, and $val(P(G), v)$ be a binary predicate symbol stating a value v of port $P(G)$. Also, let $conn(P(G), P(G'))$ be a binary predicate symbol defining the connection between $P(G)$ and $P(G')$.

As the AIG has several types of gates (i.e., *AND*, *Inv*, *PI*, and *PO*), therefore $type(G, t)$ is used to assign a gate G with a type t. As Clingo allows for encoding a constant, the logic level p is defined as constant into the program (i.e., $\#const\; p = v$, where v is a positive integer representing the logic level value). Moreover, Clingo provides an interface to represent minimum and maximum functions. Hence, the MVL *AND*, and *OR* are represented by $\#min$, and $\#max$, respectively.

For modeling the AIG graph, it is essential to define the rules for *AND*, *Inv* gates, and the connection between the ports of gates. These can be characterized as follows:

$$val(out(G), Z)\text{:- } type(G, and),$$
$$Z = \#min\{X{:}val(in1(G), X), Y{:}val(in2(G), Y)\}. \quad (3)$$
$$val(out(G), (p\text{-}1)\text{-}X)\text{:- } type(G, inv), val(in(G), X). \quad (4)$$
$$val(P2, V)\text{:- } conn(P1, P2), val(P1, V). \quad (5)$$

Eq. (3) captures the behavior of *AND* gate such that X and Y indicate the values on the ports $in1(G)$ and $in2(G)$, respectively. Also, Z corresponds to the resulting value obtained from performing minimum function on the inputs X and Y, where Z are passed to the output port $out(G)$. Similarly, *Inverter* gate is captured in Eq. (4) such that gate G takes an input value X on port $in(G)$, and outputs the resulting value on the port $out(G)$. Finally, in order to allow passing values over ports, Eq. (5) is used to define the connection between ports $P1$ and $P2$. It takes a value V on a port $P1$ and passes it to port $P2$.

To complete our encoding, ASP facts are used for representing the connections based on the structure of AIG, the values of input gates, and the type of gates. E.g., $conn(out(and10), in2(and16))$ represents the connection between the output port of gate "and10" and the second input port of gate "and16", $val(a_0, 2)$ indicates passing the value "2" on the input gate a_0, and $type(and12, and)$ identifies the gate "and12" as an *AND* gate. It is worth noting that the rules are graph-independent and can work with any AIG to model the MVL gates, while the facts depend on the structure of the AIG.

Finally, to enable verification of the circuit, it is essential to encode the sum and carry functions into ASP rules. For simplicity, we show the encoding of the MVL *OR* that is used for the MVL *XOR*. The MVL *OR* can be characterized as follows:

$$or(or, Z)\text{:- } Z = \#max\{X{:}val(P1, X), Y{:}val(P2, Y)\}. \quad (6)$$

The *XOR* can be defined from the *AND*, *Inverter*, and *OR* rules, by defining several rules, such that each rule corresponds to a basic operation. E.g., $\neg b$ is defined as a rule r and $(a \cdot \neg b)$ is defined as a rule r' from the result of r together with the value a. The sum and carry functions are defined, analogously.

Also, it is essential to relate the output gate with its expected logic function. This can be modeled as follows:

$$verify(o_i)\text{:- } s(s_i, X), val(o_i, X). \quad (7)$$
$$verify(o_{n+1})\text{:- } c(c_n, X), val(o_{n+1}, X). \quad (8)$$

As the graph has two n bit numbers and $n + 1$ outputs, it is required to relate all the sum functions s_i to the output o_i, where $0 \leq i \leq n$, while the last bit has to match the carry c_{n+1}. The sum and carry rules are depicted in Eq. (7) and Eq. (8), respectively.

We denote by $\Pi(G)$ to the program Π constructed w.r.t. the graph G. The idea behind the previous two rules is that for a given set of facts representing an input sequence of the primary inputs, the input sequence s is said to be a *Valid Input* if all outputs $verify(o_i)$ appear in the answer set of $\Pi(G)$; meaning that s satisfies all the logic functions. This leads to a characterization for the verification of the graph.

Definition 6 (Valid Graph): Let $\Pi(G)$ be a program of the graph G of size n, and $\mathcal{F}$ be a set of the sets of facts representing all possible input sequences s. Then, G is a *Valid Graph*, if for every $s \in \mathcal{F}$, we have s is a valid input. Otherwise, G is an *Invalid Graph*.

It is worth noting that the overall search space is p^n. Hence, the overall number of input sequences is $|\mathcal{F}| = p^n$.

In the following section, we propose a PFV approach for defining an upper bound of the search space.

IV. Polynomial Formal Verification of MVL Addition

In this section, we generalize the PFV approach introduced in [16] to the MVL domain. The approach divides the graph into subgraphs and stores the interleaving nodes. Thus, each subgraph is verified independently. We follow the same graph-splitting technique while generalizing the mapping functions of the information passing, and subgraph verification to the MVL domain.

A. Graph Reduction

The AIG graph can be split into subgraphs each representing an output node and its reachable nodes. I.e., the AIG G in Fig. 2(b) can be split into three subgraphs. This can be defined as follows:

Definition 7 (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:

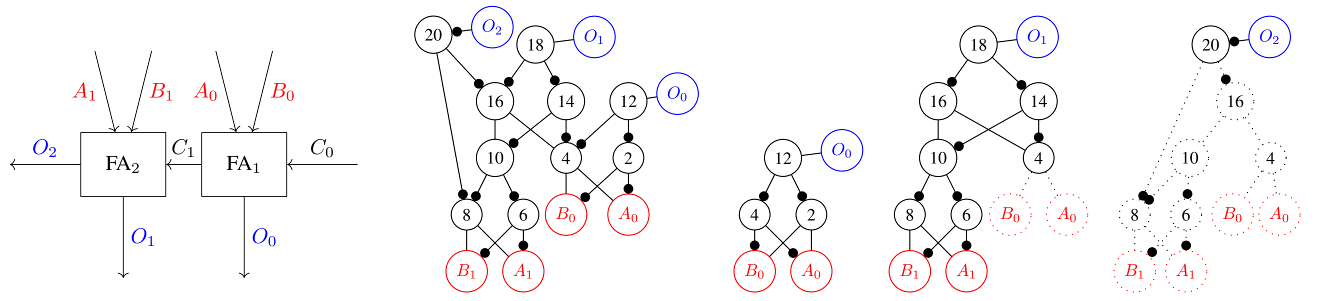


Fig. 2. The AIG G of 2-bit RCA block diagram in Fig. 2(a) and the resulting (reduced) subgraphs (G, O_0), (G, O_1) and (G, O_2) that can be obtained from the AIG graph in Fig. 2(b). The nodes highlighted in red correspond to input nodes, and those highlighted in blue correspond to output nodes. Dotted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

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

As we can see in Fig. 2, there exist some nodes that appear in several graphs. I.e., node "20" appear in (G, O_1) and (G, O_2). We denote 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 $j < i$. The set of *Out-going Nodes* CO_i is defined w.r.t. G_i as follows:

Definition 8 (Out-going Nodes): Let $G = (V, E)$ be AIG graph, and $G_i = (V_i, E_i)$ be the 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 that have a predecessor node that appears in any other graphs G_j, where $j < i$. For the graph G_1 in Fig. 2(d), we have that $CO_1 = \{20\}$, and $CI_1 = \{4\}$. As node "4" is evaluated in G_0 ($CO_0 = \{4\}$), while node "20" is evaluated in G_1 and passed to G_2 ($CI_2 = \{20\}$). Notably, $CI_0 = \emptyset$ as the reduced graph G_0 is always equivalent to (G, O_0). Similarly, $CO_n = \emptyset$.

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

In this section, we propose a method that allows for storing the set CO_i, thus it can be used in other reduced subgraphs. In simpler terms, the values of CO_i of a reduced subgraph G_i need to be stored, so that these values can be used in other subgraphs G_k, where $k > i$. These values cannot be stored as a function of the primary inputs, as this would require passing the primary inputs to other subgraphs G_k, and consequently, the last reduced subgraph would still depend on all primary inputs. Rather than storing CO_i w.r.t. the primary inputs, we store CO_i based on the computed carry function. By this means, the carry function c_i is used in the specification of the output for the subgraph.

A hash table $\mathcal{X}_i$ is used to relate the values of the out-going nodes CO_i w.r.t. the value of carry c_i. To construct the hash table $\mathcal{X}_i$, we define two mapping functions. We define a function f that maps each input sequence $s \in IN_i$ of G_i to a set of values $COUT_i$ of out-going nodes CO_i.

$$f \colon IN_i \mapsto COUT_i. \tag{9}$$

Also, we define a function g that maps each $s' \in COUT_i$ to the value of carry c_i.

$$g \colon COUT_i \mapsto [0, ..., p-1]. \tag{10}$$

Finally, the hash table $\mathcal{X}_i$ is constructed as follows:

$$\mathcal{X}_i = \{(f(s), g(f(s))) \mid s \in IN_i\}. \tag{11}$$

To illustrate this, let us consider Fig. 2(c), and assume $p = 3$. The out-going nodes $CO_0 = \{4\}$. There are three possible values for f for input sequences $s \in IN_0$. I.e., $f(0, 1) = \{0\}$, $f(1, 1) = \{1\}$, and $f(2, 2) = \{2\}$. The other input sequences are not considered as all possible values for $COUT_0$ are already covered. For function g, we have that $g(0) = 0$, $g(1) = 1$, and $g(2) = 2$. For the general case, the carry function can be replaced with one or more functions to describe the information passing over subgraphs. It should be emphasized that the number of entries of the hash table would still be bounded by the size of out-going nodes.

C. Subgraph Verification

For each subgraph G_i, we have two main tasks. The first is to check whether G_i is a valid graph (recall Definition 6), while the second involves the construction of $\mathcal{X}_i$. The inputs IN_i of G_i may contain primary inputs PI_i and in-going nodes CI_i. As CI_i might 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. Hence, a relation $\bowtie$ is used to define the relation between two table $\mathcal{X}_j$ and $\mathcal{X}_{j'}$ such that $\mathcal{X}_j \bowtie \mathcal{X}_{j'} := \{r \cup r' \mid r \in \mathcal{X}_j, r' \in \mathcal{X}_{j'}, C_j \cap C_{j'} \subseteq CI_i\}$. Thus, the resulting table $\mathcal{X}_i(CI_i)$ is defined as follows:

$$\mathcal{X}_i(CI_i) := \mathcal{X}_{i-1} \bowtie ... \bowtie \mathcal{X}_0. \tag{12}$$

Finally, the resulting table $\mathcal{X}_i(CI_i)$ is populated with the values of PI_i. In the next section, we analyze the overall time complexity of the approach.

V. Time Complexity

Let $\Pi(G_i)$ be the logic program constructed w.r.t. the reduced subgraph G_i. Then, checking the graph validity of $\Pi(G_i)$ depends on the number of inputs IN_i. This is characterized in the following theorem.

Theorem 5.1: Let G_i be the reduced subgraph, and p be a logic level. Then, $\Pi(G_i)$ is verified in time $\mathcal{O}(p^{|IN_i|})$, where IN_i is the set of all inputs of G_i.

Proof: Given a reduced subgraph G_i, and a logic level p. Then, G_i is valid, iff for every $s \in \mathcal{F}$, we have that s is a valid sequence (recall Definition 6). The set of all input sequences $\mathcal{F}$ depends on the number of inputs IN_i, where $IN_i = PI_i \cup CI_i$. Also, as each input $v \in IN_i$ may contain any value $a \in [0, ..., p-1]$. Hence, $|\mathcal{F}| = p^{|IN_i|}$, and consequently, $\Pi(G_i)$ has the search space of $|\mathcal{F}|$. Therefore, $\Pi(G_i)$ can be verified in $\mathcal{O}(p^{|IN_i|})$. ■

Since G_i may contain out-going nodes CO_i ($CO_i \neq \emptyset$), the hash table $\mathcal{X}_i$ has to be constructed. Thus, operations of the hash table take a linear time in the worst case. Hence, we assume $\mathcal{X}_i$ is constructed in a constant time. Also, as G_i may contain in-going nodes CI_i, the values of CI_i are obtained from $\mathcal{X}_i(CI_i)$. Similarly, we assume that the computation of $\mathcal{X}_i(CI_i)$ takes a constant time. This leads to a characterization of the overall time complexity of the graph G.

Theorem 5.2: Let G be a graph of a circuit of size n. Then, $\Pi(G)$ can be verified in $\mathcal{O}(n \cdot p^K)$, where n is the number of reduced subgraphs, K is the maximum size of inputs IN_i of all reduced subgraphs G_i, and p is the logic level.

Proof: Let G be the graph of a circuit with n input size. Then, the subgraphs (G, O_i) are constructed where $0 \leq i \leq n$. Also, the reduced subgraphs G_i are constructed w.r.t. (G, O_i). Due to the fact that G_i may contain in-going nodes CI_i, it is essential to compute $\mathcal{X}_i(CI_i)$ by going over all tables $\mathcal{X}_j$ that contain any node $v \in CI_i$ where $j < i$. We denote by $Comp(\mathcal{X}_i)$ to the constant time for a single access of $\mathcal{X}_i$. Therefore, the time complexity $Comp(\mathcal{X}_i(CI_i))$ is calculated as follows:

$$Comp(\mathcal{X}_i(CI_i)) := \sum_{j=0}^{i-1} Comp(\mathcal{X}_j) \qquad (13)$$

By Theorem 5.1, the program $\Pi(G_i)$ can be verified in $\mathcal{O}(p^{|IN_i|})$. Consequently, the overall time complexity of $\Pi(G)$ is computed as follows:

$$Comp(\Pi(G)) := \sum_{i=0}^{n} \mathcal{O}(p^{|IN_i|}) \qquad (14)$$

Let K be the maximum size of IN_i of all graphs G_i. Hence, by Eq. (14), $\Pi(G)$ can be verified in time $\mathcal{O}(n \cdot p^K)$. Consequently, if K is constant, then G can be verified in linear time. ■

VI. Experimental Work

In this section, we evaluate the scalability of adder circuits with constant cutwidth. We have implemented the ASP framework in Python. Notably, it can work with any circuit architecture and any logic level p. We follow the standard AIGER format [26] for the input circuit.

TABLE I
The maximum number of *input* (K), and The cutwidth (*cw*) among all reduced subgraphs for adder circuits with constant cutwidth.

Adder	**#*input* (K)**	***cw***
RCA	3	1
CSKA	8	3
CLA	11	7

A. Experimental Setup

Our evaluation includes the wall clock time and the number of timeouts in our approach (CutWidth). We use different types of bug-free adder circuits (*Ripple Carry Adder* (RCA), *Carry Skip Adder* (CSKA), and *Carry Look-ahead Adder* (CLA)) with a constant cutwidth of different sizes up to 10k input bits, and logic level p up to four-valued logic. The circuits are generated using the *ArithsGen* tool [27], and synthesized using *Yosys* [28]. The reason to choose these circuits is that it is known from [16] that these circuits exhibit a constant cutwidth.

It is worth noting that our approach verifies each reduced subgraph independently. This allows us to detect bugs in the circuit without verifying the entire circuit.

All instances are performed on Intel(R) Core(TM) i7-11370 with 3.30 GHz. We set a timeout of 1800 seconds and a limited available RAM to 16 GB per instance.

B. Experimental Results

The results displayed in Table I include the upper bound (K) of inputs (second column) and the cutwidth cw observed over all reduced subgraphs (third column), per each adder architecture (first column). The values K and cw depend on the circuit architecture (each subgraph G_i of RCA consists of two primary inputs with an in-going node representing the previous carry). The value cw represents the maximum number of out-going nodes obtained over all reduced subgraphs. In RCA, we have that $cw = 1$, as there is only one node in each reduced subgraph G_i representing the previous carry, which is then passed to the next subgraph G_{i+1}.

Moreover, Table II shows the run time of each adder architecture under different logic levels. The first column indicates the size of input bits, while the other columns capture the run time of the adder architectures per a logic level p. The run time of an instance is set to *T.O.* if the approach is not able to verify the circuit within the timeout limit.

Finally, Fig. 3 shows the run time under different logic levels for each adder architecture per input size, where the values are obtained from Table II. The dotted lines represent the run time of the adder architecture with $p = 3$, while the solid lines correspond to the adder architecture with $p = 4$. Due to the fact that these adder circuits have a constant cutwidth K, therefore, the curve of each adder circuit of different logic levels has a linear behavior. It confirms the results, we

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

#Adder	Benchmarks					
	RCA		CSKA		CLA	
	$p = 3$	$p = 4$	$p = 3$	$p = 4$	$p = 3$	$p = 4$
64	0.6	1.0	9.8	87.2	68.7	1186.4
128	1.1	1.6	20.0	185.7	144.0	2475.6
256	2.4	3.6	45.9	405.8	297.7	T.O.
512	5.1	7.7	94.4	837.1	605.1	T.O.
1024	10.5	15.4	188.0	1727.6	1228.2	T.O.
2048	21.3	32.0	379.9	T.O.	T.O.	T.O.
3072	31.9	47.6	567.8	T.O.	T.O.	T.O.
4096	38.0	57.7	763.8	T.O.	T.O.	T.O.
5120	54.0	80.4	974.6	T.O.	T.O.	T.O.
6144	66.5	97.7	1142.9	T.O.	T.O.	T.O.
7168	77.1	113.9	1337.1	T.O.	T.O.	T.O.
8192	87.6	131.8	1543.2	T.O.	T.O.	T.O.
9216	99.9	145.7	1725.0	T.O.	T.O.	T.O.
10240	110.9	163.6	1963.0	T.O.	T.O.	T.O.

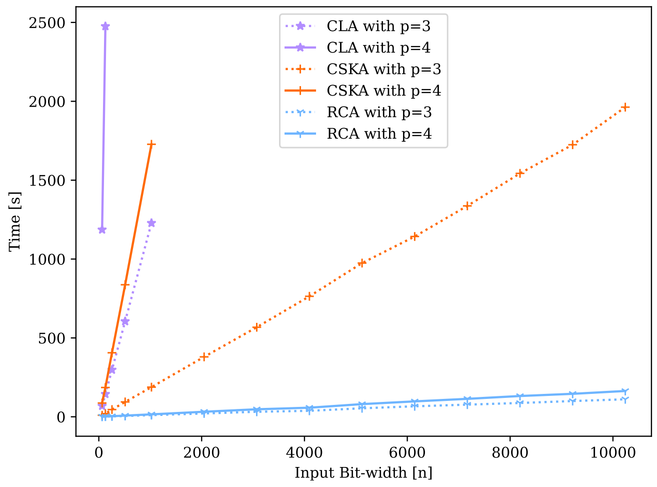


Fig. 3. Runtime graphs per adder circuit. The x-axis refers to the input bit-width, and the y-axis depicts the runtime sorted in ascending order for each circuit type individually. The dotted lines indicate the runtime obtained w.r.t. $p = 3$, while the solid lines indicate the one obtained w.r.t. $p = 4$.

obtained earlier in Theorem 5.2 that if K is constant, then the verification process is possible in linear time, irrespective of the input bitwidth n and the logic level p.

VII. Conclusion

In this paper, we have proposed a PFV approach based on cutwidth as a structural property of circuits to divide the circuit into subcircuits. Concurrently, the ASP solver is used for verifying each subcircuit and reasoning about the interleaved nodes. These interleaved nodes are stored to be used in other subcircuits. We have shown that the complexity drops to the cutwidth of the circuit and is independent of the input bitwidth. Moreover, we have shown that for a constant cutwidth, the verification process can be carried out in linear time.

As future work, we aim to examine which types of combinational MVL circuits have constant cutwidth, where the main challenge would be to adapt the hash table $\mathcal{X}$ to these circuits.

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. E. Bryant, "Binary Decision Diagrams and Beyond: Enabling techniques for formal verification," in *ICCAD*, 1995, pp. 236–243.
[4] ——, "Graph-based algorithms for Boolean function manipulation," *TC*, vol. 35, no. 8, pp. 677–691, 1986.
[5] M. Gao, J.-H. Jiang, Y. Jiang, Y. Li, A. Mishchenko, S. Sinha, T. Villa, and R. Brayton, "Optimization of multi-valued multi-level networks," in *ISMVL*, 2002, pp. 168–177.
[6] 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.
[7] 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.
[8] R. Drechsler and A. Mahzoon, "Polynomial formal verification: Ensuring correctness under resource constraints," in *ICCAD*, 2022, pp. 70:1–70:9.
[9] A. Mahzoon and R. Drechsler, "Polynomial formal verification of prefix adders," in *ATS*, 2021, pp. 85–90.
[10] ——, "Late breaking results: Polynomial formal verification of fast adders," in *DAC*, 2021, pp. 1376–1377.
[11] R. Drechsler, A. Mahzoon, and L. Weingarten, "Polynomial formal verification of arithmetic circuits," in *Proceedings of International Conference on Computational Intelligence and Data Engineering*, 2022.
[12] F. R. K. Chung, "On the cutwidth and the topological bandwidth of a tree," *SIDMA*, vol. 6, no. 2, pp. 268–277, 1985.
[13] 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.
[14] G. Brewka, T. Eiter, and M. Truszczyński, "Answer set programming at a glance," *ACM*, vol. 54, no. 12, p. 92–103, 2011.
[15] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Answer Set Solving in Practice*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning, 2012.
[16] M. Nadeem, J. Kleinekathöfer, and R. Drechsler, "Polynomial formal verification exploiting constant cutwidth," in *Proceedings of the 34th International Workshop on Rapid System Prototyping*. IEEE, 2023.
[17] P. Niemann and R. Drechsler, "Polynomial-time formal verification of adder circuits for multiple-valued logic," in *2022 IEEE 52nd International Symposium on Multiple-Valued Logic (ISMVL)*, 2022, pp. 9–14.
[18] 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.
[19] E. V. Dubrova, D. B. Gurov, and J. C. Muzio, "Full sensitivity and test generation for multiple-valued logic circuits," in *ISMVL*, 1994, pp. 284–288.
[20] H. Fujiwara, "Computational complexity of controllability/observability problems for combinational circuits," *IEEE Trans. Computers*, vol. 39, no. 6, pp. 762–767, 1990.
[21] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
[22] M. Gebser, B. Kaufmann, and T. Schaub, "Conflict-driven answer set solving: From theory to practice," *Artificial Intelligence*, 2012.
[23] I. Niemelä, "Logic programs with stable model semantics as a constraint programming paradigm," *Annals of Mathematics and Artificial Intelligence*, vol. 25, no. 3, pp. 241–273, 1999.
[24] M. Gelfond and V. Lifschitz, "Classical negation in logic programs and disjunctive databases," *New Generation Computing*, pp. 365–385, 1991.
[25] M. Gebser, R. Kaminski, A. König, and T. Schaub, "Advances in gringo series 3," in *LPNMR*, 2011, pp. 345–351.
[26] 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.
[27] J. Klhufek and V. Mrazek, "Arithsgen: Arithmetic circuit generator for hardware accelerators," in *DDECS*, 2022, pp. 44–47.
[28] C. Wolf, "Yosys open synthesis suit," available at https://github.com/YosysHQ/yosys, 2022.