Polynomial Formal Verification of Adder Circuits Using Answer Set Programming

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 GmbH
Bremen, Germany
drechsler@uni-bremen.de

***Abstract*—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. In this paper, we present our approach for PFV of arithmetic circuits. A divide and conquer approach is used to split the circuit into subgraphs. Each subgraph is verified using Answer Set Programming (ASP), a model based reasoning technique. For circuits with limited cutwidth, the verification process can be executed efficiently. Exemplarily, we use adder circuits to demonstrate the applicability of the method. Furthermore, the time complexity of the approach is analyzed and it is proven that the verification of several adder architectures is executable in linear time. Those theoretical findings are backed by experiments including different adder architectures with up to 10k bit wide inputs.**

***Index Terms*—Answer Set Programming, Polynomial Formal Verification, Logic Synthesis, Dynamic 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. matrix multiplication or AES). A fast verification approach of those circuits is crucial. Otherwise, either the task of formal verification is skipped resulting in the possibility of faulty hardware or the verification process costs the project major amounts of resources.

Many methods for the verification of arithmetic circuits were introduced in the past. SAT-based approaches create miter circuits to encode the verification as a SAT instance [1]. Decision diagrams [2] can be used for the verification by creating the diagrams for representing the output function of circuits. Those diagrams can then be compared to the specification. Because in practice many designs could not be verified with those approaches, alternatives were introduced. Theorem proving uses a manual approach to enable the verification of complex circuits [3]. Symbolic computer algebra (SCA) proved to be well suited for multiplier verification [4]. The computational complexity of those approaches is exponential for the most part. The SAT problem is NP-hard, decision diagrams can get exponentially big and polynomials used in SCA can reach exponential size. Theorem proving on the other hand needs time consuming manual work. To nevertheless ensure efficient verification, PFV methods guarantee polynomial runtime for a specific class of circuits.

Due to the limitations of the SAT encoding of circuits, it is essential to enable more compact encoding for the logic functions of circuits. *Answer Set Programming* (ASP) [5]–[7] is a declarative modeling and problem solving framework oriented towards difficult (NP-hard) search problems, where these search problems are reduced to computing answer sets (stable models). It is well-known in the area of knowledge representation and non-monotonic reasoning [8]. Unlike SAT, ASP offers a compact representation of problems and enables a straightforward way of modeling logic functions representing a circuit.

There are many types of arithmetic circuits on which the introduced methods could be applied. One common arithmetic circuit is the adder circuit. Adder circuits are used in CPUs and more complex arithmetic circuits (i.e. floating point arithmetic). Many different adder architectures are known. The verification of those adders was proven to be possible in polynomial time for conditional sum adders, carry look ahead adders, conditional sum adders as well as prefix adders [9]–[11].

In this paper we introduce a new approach for the verification of arithmetic circuits and apply it to the verification of adders. The approach splits the netlist into subcircuits which are connected through cone nodes. Accordingly, the approach can be described as a divide and conquer approach. Each subcircuit is verified independently and the information passed between the subcircuits is saved. The subcircuit verification is performed by an ASP solver. Different techniques (i.e. SAT solvers or decision diagrams) could be applied, but in this paper we focus on the introduction of the approach in combination with ASP. Moreover, to the best of our knowledge we are the first to apply ASP on PFV of arithmetic circuits. The complexity of the verification is mainly determined by the cutwidth between the subcircuits. This is particularly interesting, because it aligns with results known from automatic test pattern generation [12]. It is proven that the approach is able to verify adder circuits with constant cutwidth in linear time. This is backed by the experiments which examine the performance

This work was supported by the German Research Foundation (DFG) within the Project *PLiM* (DR 287/35-1) and the Reinhart Koselleck Project *PolyVer* (DR 287/36-1).

of the approach for different input sizes and architectures.

In Section II we introduce the concept of ASP as well as the relevant adder circuits. Subsequently the modeling of circuits in ASP is described in Section III. Our approach for polynomial formal verification 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. Answer Set Programming

In this section we introduce the basic idea and concepts behind ASP. We follow standard definitions of ASP [13], [14].

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. It is very related to the one pursued in SAT, where problems are encoded as propositional theories, whose models represent solutions of a given problem. In ASP, the logic program is built from basic notions, that correspond to the language of first-order predicate calculus.

A key concept in ASP are *constants*, which represent individuals of the domain, denoted by a lowercase starting letter or with a natural number, like a, b. *Variables* represent individual variables and are denoted by an uppercase starting letter, like X, Y. Moreover, one can use *functional terms* combining variables, constants and *functional symbols*, like $out(a)$, where out is a unary function symbol. Finally, *predicate terms* are used to relate all variables, constants and functional terms together through *atoms*, like $conn(out(a), in(b))$. An atom a is said to be a *ground atom*, if a does not contain any variable. Otherwise, it is a *non-ground atom*. A ground atom a' can be obtained from a non-ground atom a by replacing variables with constants. This process is called *grounding*. Therefore, ground atoms can be considered as ordinary propositions. i.e, ground atoms can be assigned to a truth value true ($\top$) or false ($\bot$). Atoms are the basic building blocks of logic programs.

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

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

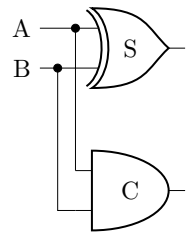
where $l \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom, where $1 \leq i \leq n$. Atoms $a_1, ..., a_m$ are called positive atoms, while atoms $\neg a_{m+1}, ..., \neg a_n$ are called negative atoms. Let $H_r := \{a_1, ..., a_l\}$ and $B_r := \{a_{l+1}, ..., a_n\}$. Also, let $B_r^+ := \{a_{l+1}, ..., a_m\}$ and $B_r^- := \{a_{m+1}, ..., a_n\}$.

Definition 2 (Fact): Let r be a rule of Π. Then, r is said to be a *fact*, if and only if $B_r := \phi$.

Definition 3 (Atoms of a Rule): Let $r \in \Pi$ be a rule of Π. Then, $at(r)$ is a set of atoms such that:

$$at(r) := H_r \cup B_r^+ \cup B_r^-$$

Definition 4 (Positive Program): Let Π be a logic program. Then, Π is said to be *positive* if $B_r^- = \phi$, for all $r \in \Pi$.
The answer set (stable model) semantics is defined w.r.t. a positive program as follows.



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.

Definition 5 (Answer Set of a Positive Program): Let Π be a positive program and $Q \in at(\Pi)$ be a set of atoms. Then, Q is said to be a *model* of Π if for any $r \in \Pi$, we have $H_r \in Q$ whenever $B_r^+ \subseteq Q$. We refer by $Cn(\Pi)$ to a smallest set of atoms, which is a model of the positive program Π. The set $Cn(\Pi)$ of atoms is the answer set of the positive program Π.

For the general case, answer sets are defined in terms of a *reduct* [15] of a program Π relative to a set Q of atoms.

Definition 6 (Gelfond-Lifschitz Reduct): Let Π be a program and Q be a set of atoms. Then, the reduct Π^Q of a program Π is defined as follows:

$$\Pi^Q := \{H(r) \leftarrow B_r^+ \mid r \in \Pi, B_r^- \cap Q = \phi\} \qquad (2)$$

To illustrate (2), a reduct Π^Q of a program Π under a set Q of atoms is the program obtained by first removing all rules r with $B_r^- \cap Q = \phi$ 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 7 (Answer Set): Let Π be a logic program and Q be a set of atoms. 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 $AS(\Pi) := \{Q \subseteq at(\Pi) \mid Cn(\Pi^Q) = Q\}$.

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 atoms that

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 atoms $Cn(\Pi^Q)$ per a set Q of atoms.

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

satisfy the condition of being an answer set ($Cn(\Pi^Q) = Q$). In Table I, some possible set of atoms Q are removed, as they are not satisfied by the answer set condition.

B. Adder Function

Let a, b be two inputs with size n bits, and $carry_{-1}$ be a carry bit, that represents 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.

$$sum_i := a_i \oplus b_i \oplus carry_{i-1} \tag{3}$$

The carry bits can be characterized as follows.

$$carry_i := (a_i \wedge b_i) \vee (carry_{i-1} \wedge (a_i \oplus b_i)) \tag{4}$$

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 $carry_n$.

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) [16] representations that are well-known in synthesis of logic functions.

III. Circuit Modeling Using ASP

We follow the same modeling of an AIG graph representation of a circuit that was introduced in [17] with some extensions. 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 8 (AIG Graph): Let $G = (V, E)$ be a directed acyclic graph such that:

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

In order to enable modeling, the behavior of gates is modeled using ASP rules. Also, we introduce ports for each gate to enable connections of gates. These ports provide a mechanism to handle passing values between a gate and its connections. Moreover, gate behavior is defined based on values on their ports. Thus, we will make use of the following predicate and function symbols for modeling of an AIG:

1) A unary function symbol $P(G)$ representing a port of gate G.
2) A binary predicate symbol $val(P(G), v)$ stating the value v on a port P of gate G.
3) A binary predicate symbol $conn(P(G), P'(G'))$ defining the connection between P of G and P' of G'.

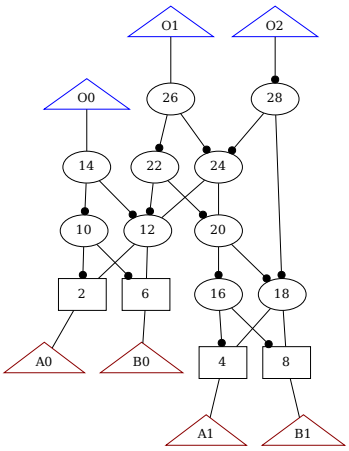


Fig. 2. Example of an AIG graph of 2-bit RCA.

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. 2 illustrates an AIG for a ripple carry adder with 2-bit inputs. For the encoding of the circuit as an ASP logic program we rely on the input language of the ASP tool *clingo* [18], [19], which is an extended version of *prolog* [20].

To convert facts and rules that are introduced in Eq. (1) 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 "^", respectively. Those operations are used to describe the behavior of a gate. 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:

$$val(out(G), X\&Y) : -type(G, and), val(in1(G), X), \\ val(in2(G), Y). \tag{5}$$

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:

$$val(out(G), 1\hat{}X) : -type(G, inverter), \\ val(in(G), X). \tag{6}$$

In Eq. (6), 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:

$$val(P2, V) : -conn(P1, P2), val(P1, V). \quad (7)$$

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., $val(a_0, 1)$. The primary outputs are represented analogously, except that their values are observed from the circuit. Informally, by Eq. (7), 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. (3) and Eq. (4). Therefore, it is essential to encode sum and carry functions into clingo. The sum function can be characterized as follows:

$$sum(sum_i, V) : -val(a_i, A), val(b_i, B), \\ carry(carry_{i-1}, C), V = A \oplus B \oplus C. \quad (8)$$

The carry function can be characterized as follows.

$$carry(sum_i, V) : -val(a_i, A), val(b_i, B), \\ carry(carry_{i-1}, C), V = (A\&B)?(C\&(A \oplus B)). \quad (9)$$

Equations (5), (6), (7), (8), and (9) are very general and can work independently of the circuit architecture. 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 $conn(out(inv1), in1(and2))$. For simplicity, we show only the facts for gates "A0", "B0" and "2" of Fig. 2, together with their connections, and they are summarized as follows.

$$type(a_0, cirIn).type(b_0, cirIn).type(and2, and). \\ type(inv1, inverter).type(inv2, inverter). \\ conn(a_0, in(inv1)).conn(b_0, in(inv2)). \\ conn(out(inv1), in1(and2)).conn(out(inv2), in2(and2)).$$

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. (8) and Eq. (9)). Thus, we introduce one clingo rule per output bit to reach the desired behavior. Considering Fig. 2, the clingo rules for the verification of all outputs can be summarized as follows:

$$\textit{verify}(o_0) : -sum(sum_0, X), val(o_0, Y), X = Y. \\ \textit{verify}(o_1) : -sum(sum_1, X), val(o_1, Y), X = Y. \\ \textit{verify}(o_2) : -carry(carry_1, X), val(o_2, Y), X = Y. \quad (10)$$

The idea behind Eq. (10) 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 *verify*(o_i) appears in the answer set of the program. This can be formulated as follows.

Definition 9 (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 9, if the input sequence is correct, all values of the outputs are equal to their corresponding logic function and consequently, their verification atoms *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 10 (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 polynomial 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 [12] 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 the terminology of graphs and trees [21].

A. Graph Unraveling and Reduction

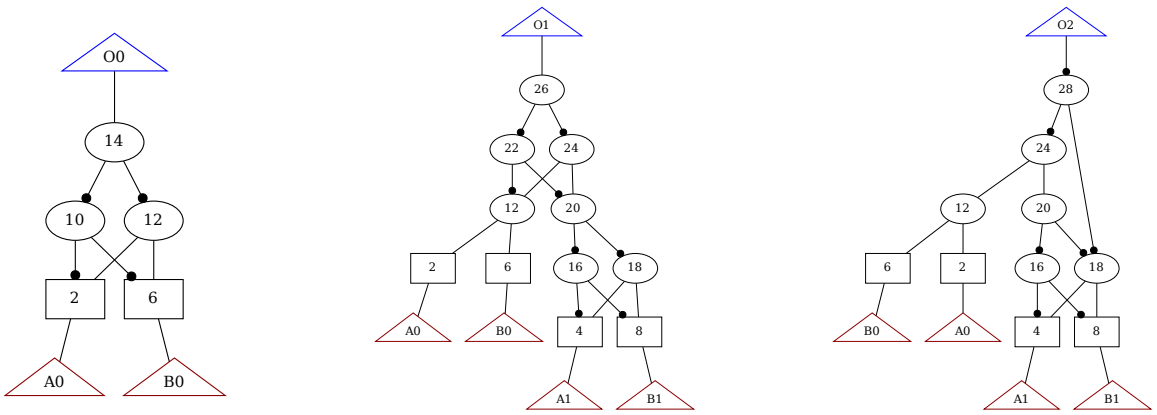
Given an AIG graph $G = (V, E)$ and a node $v \in V$, a subgraph (G, v) can be constructed as follows.

Definition 11 (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:

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

In graph theory, the cutwidth [22] 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



(a) (G, O_0). (b) (G, O_1). (c) (G, O_2).

Fig. 3. The resulting subgraphs (G, O_0), (G, O_1) and (G, O_2) that can be obtained from AIG graph in Fig. 2.

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. 3. 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 12 (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 10 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 13 (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 13, 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 the number representing 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 [23] 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 13 yields a characterization of *k-bounded circuit* in terms of the graph.

Definition 14 (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

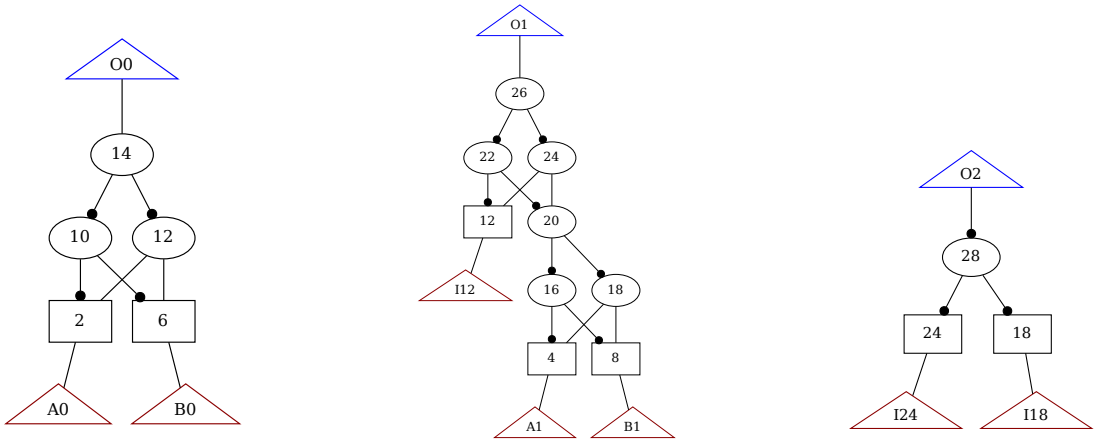
As we can see in Fig. 4, each cone node v is evaluated only once in one of the subgraphs. Also, any other graph that uses v as an input, takes the value of v from the graph in which it is evaluated. Hence, the cone node values of reduced subgraphs must be stored, so their values can be used in other 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. 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 $C(G_i)$.

$$f \colon IN_i \mapsto COUT_i \tag{11}$$

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

$$g \colon COUT_i \mapsto [0, 1] \tag{12}$$



(a) $R(G, O_0)$. (b) $R(G, O_1)$. (c) $R(G, O_2)$.

Fig. 4. The resulting reduced subgraphs $R(G, O_0)$, $R(G, O_1)$ and $R(G, O_2)$ obtained from reducing all subgraphs in Fig. 3. The nodes highlighted in red correspond to input nodes, and those highlighted in blue correspond to output nodes.

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 13 defines the hash table accordingly.

$$\mathcal{T}_i = \{(f(s), g(f(s)) \mid s \in IN_i\} \tag{13}$$

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 $\mathcal{X}_i(CIN)$ to the resulting table containing the values of CIN and defined as follows.

$$\mathcal{X}_i(CIN) := \mathcal{T}_{i-1} \bowtie ... \bowtie \mathcal{T}_0. \tag{14}$$

Finally, every $r \in \mathcal{X}_i(CIN)$ 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 10), 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 10, 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 $X_i(CIN)$ (recall Eq. (14)). We assume that $\mathcal{X}_i(CIN)$ 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 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, $\mathcal{X}_i(CIN)$ can be computed in constant time $P(\mathcal{X}_i(CIN))$.

Finally, the overall time required to verify $\Pi(G)$ 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 11, where $0 \leq i \leq n$. Also, a reduced subgraph R_{G_i} can be obtained from (G, v_i) by applying Definition 13. By Definition 13, 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 $CIN_0 = \phi$. 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 $\mathcal{X}_i(CIN)$ (Equation 14) where $0 < i \leq n$. This is due to the fact that for all R_{G_i}, where $i > 0$, and $CIN_i \neq \phi$. Since $\mathcal{X}_i(CIN)$ 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 $\mathcal{X}_i(CIN)$ can be calculated as follows:

$$\textit{Complexity}(\mathcal{X}_i(CIN)) := \sum_{j=0}^{i-1} P_j \tag{15}$$

We refer by $P(\mathcal{X}_i(CIN))$ 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 $\mathcal{O}(2^{|IN_i|} + P(\mathcal{X}_i(CIN))) = \mathcal{O}(2^{|IN_i|})$, where $0 < i \leq n$. The overall time complexity for verifying $\Pi(G)$ can be calculated as follows:

$$\textit{Complexity}(\Pi(G)) := \sum_{i=0}^{n} \mathcal{O}(2^{|IN_i|}) \tag{16}$$

Moreover, by Definition 14, 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 16 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 [24]. Then, the input circuit is evaluated to see whether it is a valid or an invalid circuit.

A. Experimental Setup

Measure and Resources. We mainly compare the wall clock time and the number of timeouts. We set a timeout of 1500 seconds and a limited available RAM to 8 GB per circuit instance.

Benchmark Instances. We use three types of bug-free adder circuits of different sizes (Ripple Carry Adder (RCA), Carry Skip Adder (CSKA), and Carry Look Ahead (CLA)). These circuits are generated using the ArithsGen tool [25], where the design is synthesized using Yosys [26].

Benchmark Hardware. All instances are performed on Intel(R) Core(TM) i7-11370 with 3.30 GHz and 16 GB of memory.

TABLE II
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	4	7
CSKA	9	15
CLA	11	18

TABLE III
Run time of verifying adder circuit (seconds).

Size	Benchmarks		
	RCA	**CSKA**	**CLA**
1024	8.79	49.83	189.58
2048	21.77	105.09	397.86
3072	39.63	167.93	611.72
4096	62.30	233.20	871.86
5120	89.06	308.79	1312.50
6144	124.32	390.16	T.O.
7168	161.21	472.60	T.O.
8192	199.01	572.60	T.O.
9216	248.80	672.24	T.O.
10240	300.32	759.22	T.O.

B. Experimental Results

Table II shows the results of the upper bound of inputs (as shown in the second column) and the maximum number of and gates appearing in reduced subgraphs (as shown in the third column) for adder architectures (as shown in the first column) during the verification process. We can observe that the upper bound is circuit dependent. 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. It is worth noting that each reduced subgraph is also bounded in terms of the number of *and* gates.

Table III compares the run time of each adder architecture w.r.t. different input size, where the input size is represented in the first column, and the second, third, and fourth columns represent the run time of RCA, CSKA, and CLA, respectively. If the ASP framework was not able to solve the instance within the timeout limit, then the run time of this instance is set to *T.O.* (e.g., a CLA of "6144" input size) as shown in Table III. It shows that the ASP-based verification has a good time performance for different types of adder circuits. Also, it confirms the scalability of the proposed approach. Since the proposed approach relies on the upper bound *K*, this explains why one circuit scales faster in terms of run time than another one. More precisely, the run time of the CLA scales faster than the run time of the RCA, as $K = 11$ for the CLA,

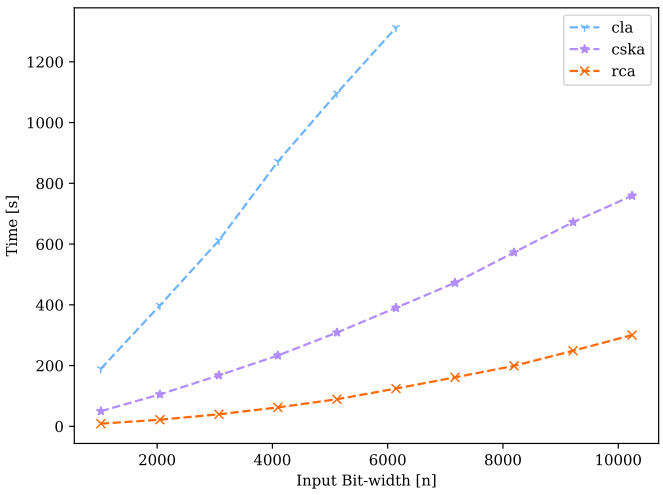


Fig. 5. Run time graphs per adder circuit. The x-axis refers to the input bit-width, and the y-axis depicts the run time sorted in ascending order for each circuit type individually.

while $K = 4$ for the RCA. Analogously, the CSKA scales faster than the RCA.

Moreover, Fig. 5 shows the run time of each adder architecture per input size, where the run time per input size is shown in Table III. Hence, the curve of each adder architecture is a linear curve. Therefore, it confirms the correctness of the calculated complexity bound obtained from Theorem 5.2 in Section V.

VII. Conclusion

In this paper, we have proposed a new polynomial verification approach that relies on the cutwidth of a netlist, where ASP was used to verify each subcircuit 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. 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," *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 112–136, 2001.

[3] J. Harrison, "Theorem proving for verification (invited tutorial)," in *Computer Aided Verification*, 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 *International Conference on Computer-Aided Design*, 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, proceedings of the 1st intl. asp'01 workshop," 2001.

[8] F. Harmelen, V. Lifschitz, and B. Porter, "The handbook of knowledge representation," p. 1034, 2007.

[9] R. Drechsler, "PolyAdd: Polynomial formal verification of adder circuits," in *IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2021, pp. 99–104.

[10] A. Mahzoon and R. Drechsler, "Polynomial formal verification of area-efficient and fast adders," in *Reed-Muller Workshop*, 2021.

[11] ——, "Polynomial formal verification of prefix adders," in *Asian Test Symp.*, 2021, pp. 85–90.

[12] 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.

[13] V. Marek and M. Truszczynski, "Stable models and an alternative logic programming paradigm," 1998.

[14] 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.

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

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

[17] F. Wotawa and D. Kaufmann, "Model-based reasoning using answer set programming," *Applied Intelligence*, vol. 52, pp. 1–19, 2022.

[18] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Clingo = asp + control: Preliminary report," May 2014.

[19] M. Gebser, R. Kaminski, A. König, and T. Schaub, "Advances in gringo series 3," in *Logic Programming and Nonmonotonic Reasoning*, 2011, pp. 345–351.

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

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

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

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

[24] 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.

[25] J. Klhufek and V. Mrazek, "ArithsGen: Arithmetics circuit generator for hw accelerators," in *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2022, p. 4.

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