TPLP: Page 1–18.

doi:10.1017/S1471068425100288

Fine-Grained Timing Analysis of Digital Integrated Circuits in Answer Set Programming[*]

ALESSANDRO BERTAGNON
Department of Environmental and Prevention Sciences, University of Ferrara, Ferrara, Italy
(*e-mail:* alessandro.bertagnon@unife.it)

MARCELLO DALPASSO
DEI - University of Padova, Padova, Italy
(*e-mail:* marcello.dalpasso@unipd.it)

MICHELE FAVALLI and MARCO GAVANELLI
Department of Engineering, University of Ferrara, Ferrara, Italy
(*e-mail:* michele.favalli@unife.it, marco.gavanelli@unife.it)

submitted 16 July 2025; revised 28 July 2025; accepted 29 July 2025

Abstract

In the design of integrated circuits, one critical metric is the maximum delay introduced by combinational modules within the circuit. This delay is crucial because it represents the time required to perform a computation: in an Arithmetic Logic Unit, it represents the maximum time taken by the circuit to perform an arithmetic operation. When such a circuit is part of a larger, synchronous system, like a CPU, the maximum delay directly impacts the maximum clock frequency of the entire system. Typically, hardware designers use static timing analysis to compute an upper bound of the maximum delay because it can be determined in polynomial time. However, relying on this upper bound can lead to suboptimal processor speeds, thereby missing performance opportunities. In this work, we tackle the challenging task of computing the actual maximum delay, rather than an approximate value. Since the problem is computationally hard, we model it in answer set programming (ASP), a logic language featuring extremely efficient solvers. We propose non-trivial encodings of the problem into ASP. Experimental results show that ASP is a viable solution to address complex problems in hardware design.

KEYWORDS: answer set programming applications, hardware design, answer set programming encodings, integrated circuit maximum delay

1 Introduction

In the design of integrated circuits, one critical metric is the maximum delay introduced by combinational modules within the circuit (Hitchcock 1982; Agrawal 1982; Kundu 1994;

[*] Alessandro Bertagnon and Marco Gavanelli are members of the Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

Gharaybeh *et al.* 1998; Lu *et al.* 2004; Chadha and Bhasker 2009; Andres *et al.* 2013a, 2013b; Wang and Robinson 2019; Shi *et al.* 2024). For example, in digital synchronous circuits, the maximum delay in the combinational (i.e., acyclic) logic blocks is necessary to determine the clock period. For instance, an Arithmetic Logic Unit (ALU) is a crucial component of a CPU; it is a combinational circuit (its output depends only on the inputs, and it does not have an internal memory), but its speed influences the overall speed of the CPU, which is a synchronous sequential circuit. When designing an ALU, the designer must determine the time required to produce the final output after a new input is provided to the circuit; the maximum possible delay influences the maximum clock rate the CPU can run at. This step is first performed at early design steps using simplified delay models for gates in order to drive possible circuit optimizations, and finally it is performed after the physical design using more complex delay models that use accurate electrical level information. For simplicity, we will consider the basic case, although the proposed approach can be extended to more accurate delay models.

An approximate approach to maximum delay computation is given by static timing analysis (STA) (Hitchcock 1982; Agrawal 1982; Chadha and Bhasker 2009), which computes this delay as the longest path in a directed acyclic graph. The circuit is traversed from primary inputs (PIs) to primary outputs (POs); the static arrival time for the output of each gate is the sum of the maximum arrival time of its inputs plus the gate propagation delay. This method is static because it does not consider the actual logic values within the circuit. Therefore, it provides possibly pessimistic results since there might exist no input configuration that propagates transitions through the computed longest paths.

As an example, consider the circuit in Figure 1, where the delay introduced by each gate is represented by a number inside it (in arbitrary time units). In such a circuit, the maximum delay computed by STA is 12, with the path *b*-*f*-*h*-*i*-*k*-*l*-*n*. Paths featuring the maximum delay are called critical paths (Kundu 1994). However, no sequence of input vectors exists that makes a transition propagate through such a path. In fact, if signal o is 0 (false), then h is always 0, independently of b, while if p is 1 (true), the output l is always 0, independently from the path highlighted in red. So, the only possibility for a signal from b to influence the output n would be that $o = 1$ and $p = 0$, which is impossible since $o = c \wedge d$ and $p = c \vee d$. The maximum delay of such a circuit, which can be computed with more accurate approaches and by means of the proposed method, is 10.

To refine such results with the aim to meet the aggressive timing requirements of today's circuits, several approaches consider circuit paths and try to sensitize them to prove that transitions can propagate through them. If this operation is possible, the path is said to be true; otherwise, the path is said to be false (such as the path indicated in Figure 1 because it cannot contribute to the maximum circuit delay (Gharaybeh *et al.* 1998). For computational reasons, only the longest paths are typically considered (Lu *et al.* 2004). The way in which a path is sensitized determines the quality of the results achievable by such timing verification techniques.

For example, Andres *et al.* (2013a, 2013b) use answer set programming (ASP) to compute the maximum delay in a circuit. They search for a sequence of two input vectors such that when the circuit passes from the first bit vector to the second, a transition is

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

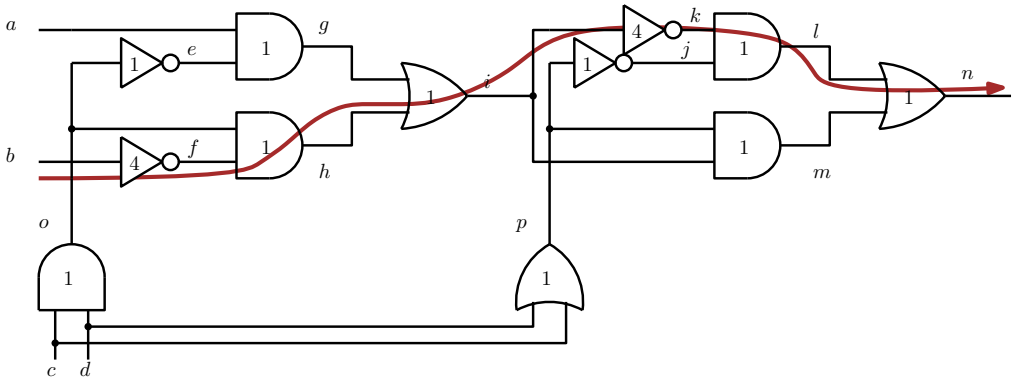

Fig. 1. Example of circuit featuring a false path that leads to pessimism in STA.

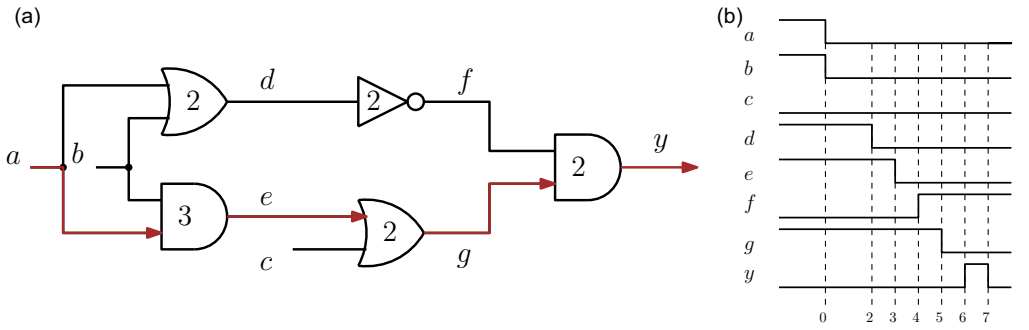

Fig. 2. (a) Example circuit and (b) corresponding signal waveforms.

propagated through the longest sensitizable path. A path is sensitizable if all the gates in the path flip (change their output from 1 to 0 or vice versa) when the input changes from the first input vector to the second.

Consider, for instance, the circuit in Figure 2a. The path *a*-*d*-*f*-*y* can be sensitized, in fact if we take as first input vector $[a=0, b=0, c=1]$ and as second $[a=1, b=1, c=0]$, all the gates in the path switch their output value (d, f and y change value). The total delay on this path is the sum of the delays in the path: $2+2+2=6$.

Instead, the path *a*-*e*-*g*-*y* cannot be statically sensitized, in fact in order for the input a to change the output e of the first AND gate, it is required that $b=1$, and for the same reason $c=0$ and $f=1$. It is easy to find that $b=1 \rightarrow f=0$, thus preventing a transition on a to reach the output. Such path has delay = 7 and cannot be found with path sensitization. Unfortunately, although the longest sensitizable path has length 6, this does not mean that after 6 time units, the final value has been correctly computed, or that there cannot be unpleasant effects happening after such time.

Consider, for example, the switch from $[a=1, b=1, c=0]$ to $[a=0, b=0, c=0]$; the signals' waveforms are shown in Figure 2b. At time 0, a and b switch from level 1 to 0; 2 time units later, d flips to 0, and at time 4, signal f goes to 1. Signal e changes at time 3, causing g to flip to 0 at time 5. Note that between time 4 and 5, both f and g have value 1: this makes the final AND gate switch its output from 0 to 1 in instant 6. This is

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

not the final output value: in fact at time 5, the signal g goes to 0, making the *out* signal go to its final level only at time 7. The temporary switching of a signal is called in the literature a *hazard*; the circuit in Figure 2a has a 1-*sized hazard* of the y signal, since it lasts 1 time unit.

In the next section, we propose a model of a circuit that also takes into consideration hazards when computing the maximum delay.

2 Problem description

Unlike path-based approaches, which constitute a large majority in the field of timing verification to improve STA results, our approach does not explicitly consider paths. Instead, it directly models the dynamic conditions occurring within the circuit for the possible input configurations.

Let us consider a combinational circuit with a set $\mathbf{I}$ of input signals, where the number of inputs is denoted by $n_i = |\mathbf{I}|$, and a set $\mathbf{O}$ of output signals, with $n_o = |\mathbf{O}|$.

To compute the maximum possible delay of the circuit, we study its behavior between two arbitrary input vectors, $\mathbf{V}_1$ and $\mathbf{V}_2 \in \{0,1\}^{n_i}$, which are applied one after the other to the input signals.

The state of each signal s in the circuit is represented by a 4-tuple $\langle v_1^s, v_2^s, e^s, l^s \rangle$, where 1) $v_1^s \in \{0,1\}$ is the logic value to which s stabilizes when the first input vector is applied; 2) $v_2^s \in \{0,1\}$ denotes the value to which s eventually stabilizes after applying the second input vector; 3) $e^s \in [-1, T]$ denotes the earliest time at which a transition away from v_1^s may occur; 4) $l^s \in [-1, T]$ denotes the latest time by which s stabilizes to v_2^s. T denotes a value much larger than the expected maximal delay of the considered combinational circuit.

The intuitive meaning is that initially the first input vector is applied, and the signal s stabilizes to the logical value v_1^s. When the second input vector is introduced, the signal does not change immediately, due to propagation delays in the circuit. It remains at v_1^s until the early arrival time e^s, after which it might have spurious variations. Eventually, by time l^s (called the latest stabilization time), it becomes stable to the final value v_2^s, which depends only on the second input vector.

The objective is to determine the worst-case delay, defined as the maximum stabilization time across all outputs, over all possible pairs of input vectors $\mathbf{V}_1$ and $\mathbf{V}_2$:

$$\max_{\mathbf{V}_1, \mathbf{V}_2 \in \{0,1\}^{n_i}} \{ l^o \mid o \in \mathbf{O} \}.$$

The gate output state is computed as a function of gate input states. Let us consider a NAND gate with output signal y and a, b as inputs. The input/output relationships for the two considered test vectors are described as $v_i^y = \neg(v_i^a \wedge v_i^b)$ for each of the two input vectors (represented by the index $i \in \{1,2\}$).

The arrival times of the gate output are a function of the input logic values and of the arrival times of the gate inputs. Let us instantiate the computation of e^y and l^y in a gate delay model featuring the propagation delay d as the only parameter; as already said, more complex delay models can be easily accounted for.

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

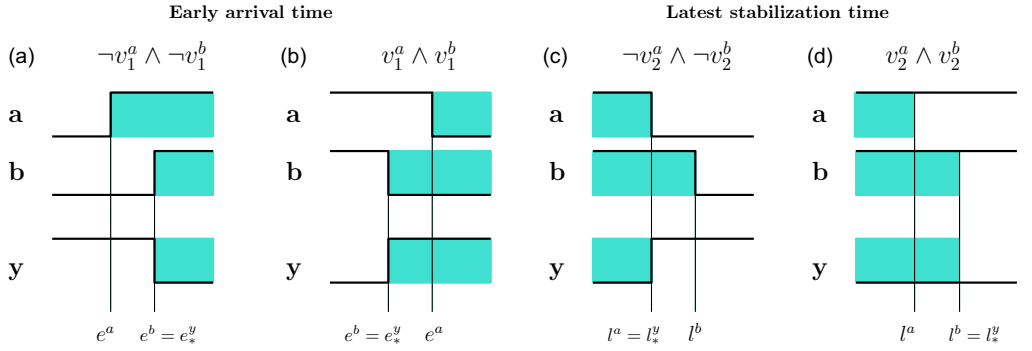

Fig. 3. Examples of the application of equation (1). The shaded areas denote the interval between the early and the latest stabilization time. In such time regions, the signal value is not determined because we also account for multiple hazards. Each case shows the early arrival time (left) and the latest stabilization time (right) of the output, considering inputs with the controlling starting value 0 (a) and 1 (b), or the controlling final value 0 (c) and 1 (d).

To this purpose, we first introduce two auxiliary variables e_*^y and l_*^y that intuitively represent the time in which the gate may start commuting (e_*^y) and that in which the output becomes stable (l_*^y) without considering the delay introduced by the gate itself.

$$v_1^a \wedge v_1^b \rightarrow e_*^y = \min\{e^a, e^b\} \tag{1a}$$

$$\neg v_1^a \wedge \neg v_1^b \rightarrow e_*^y = \max\{e^a, e^b\} \tag{1b}$$

$$v_1^a \wedge \neg v_1^b \rightarrow e_*^y = e^b \tag{1c}$$

$$\neg v_1^a \wedge v_1^b \rightarrow e_*^y = e^a \tag{1d}$$

$$v_2^a \wedge v_2^b \rightarrow l_*^y = \max\{l^a, l^b\} \tag{1e}$$

$$\neg v_2^a \wedge \neg v_2^b \rightarrow l_*^y = \min\{l^a, l^b\} \tag{1f}$$

$$\neg v_2^a \wedge v_2^b \rightarrow l_*^y = l^a \tag{1g}$$

$$v_2^a \wedge \neg v_2^b \rightarrow l_*^y = l^b \tag{1h}$$

For instance, equation (1a) considers the case in which the initial values of a and b are both 1. Since the NAND gate has output 1 whenever one of its inputs is 0, as soon as the first of its inputs switches to 0, the output will leave its initial 0 value, as shown in Figure 3b. Value 0 is called a *controlling value* for the NAND gate, since as soon as one of its input takes the controlling value, the other inputs are irrelevant for the output. Other gate types have a controlling value; we define this concept formally, on a generic logic gate indicated with the symbol $\odot$.

Definition 1
(Controlling value). *The logic gate* $\odot$ *has a controlling value* c *iff* $c \odot 1 = c \odot 0 = 1 \odot c = 0 \odot c$; *in such a case* $\neg c$ *is called non-controlling value.*

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

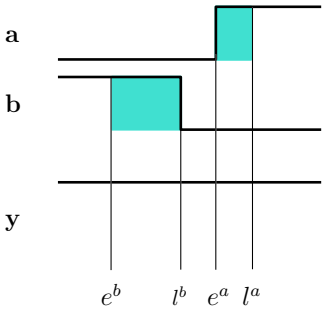

Fig. 4. Example of timing waveforms featuring a potential hazard which, however, does not occur because of the relative timing of signals.

Conversely, equation (1b) considers the situation in which both inputs have the controlling value 0. Therefore (Figure 3a), y will switch only when both inputs have switched. Equations (1c) and (1d) handle the cases in which an input takes the controlling value and the other the non-controlling value: the output will switch when the input at the controlling value switches.

Equations (1e)–(1h) are used to compute the latest stabilization time of the output signal y. Equation (1e) takes care of the case in which a and b have 1 (non-controlling) as final values. The output stabilizes to 0 only when both the inputs are stable (Figure 3d). Conversely, in equation (1f), both inputs have a controlling final value 0. Therefore, y will stabilize to its final value 1 as soon as one of such inputs has stabilized (Figure 3c). Equations (1g) and (1h), instead, feature an input with a final controlling value and the other one with a non-controlling value; therefore, the output will stabilize to its final value only when the input at the controlling value has stabilized.

In order to compute the times e^y and l^y, we have to consider that the time interval $[e^y_*, l^y_*]$ might be of zero or negative length. For example, consider the case featuring a rising transition on a ($v^a_1 = 0$ and $v^a_2 = 1$) and a falling one on b ($v^b_1 = 1$ and $v^b_2 = 0$), with the following timing: $e^a = 4$, $l^a = 5$ and $e^b = 1$, $l^b = 3$, also depicted in Figure 4. It is evident from Figure 4 that there is no instant in which both inputs are 1 at the same time, which implies that the output y remains stable at 1, meaning that no hazard may be present at the output. Note that equations (1d) and (1g) produce $e^y_* = e^a_* = 4$ and $l^y_* = l^b_* = 3$, respectively. The apparent inconsistency $l^y_* < e^y_*$ signals that there is no transition on y, which can be thought as if y reaches its final value at time $-\infty$, and that the initial value is maintained until time $+\infty$. Steady states of a signal s (hazard and transition free within the two considered input vectors) are encoded by $e^s = T$, and $l^s = -1$. Here, T is a value larger than the maximum delay computed by STA, effectively representing an infinite early arrival time (the signal never switches from v^s_1), while $l^s = -1$ signifies that the signal has already stabilized to v^s_2 when the first input vector is applied). We detect this situation when the following variable η_y is true:

$$\eta_y \leftrightarrow (v^y_1 = v^y_2) \wedge (l^y_* - e^y_* \leq 0) \ . \tag{2}$$

Such a variable holds true if and only if there is a potential hazard that does not appear because of circuit delays.

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

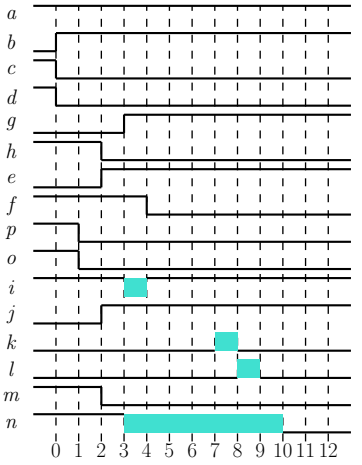

Fig. 5. Waveforms computed in the example circuit of Figure 1, in the case of maximum delay. The colored areas represent timings in which the value of the signals is undetermined. Waveforms automatically drawn with ASPECT (Bertagnon and Gavanelli, 2024a).

The gate propagation delay can be added in all cases that are different from a constant gate output. This is made by adding the following constraints:

$$(e^y_* \neq T) \wedge \neg\eta_y \ \rightarrow (e^y = e^y_* + d) \tag{3a}$$

$$(l^y_* \neq -1) \wedge \neg\eta_y \ \rightarrow (l^y = l^y_* + d) \tag{3b}$$

$$(e^y_* = T) \vee \eta_y \ \rightarrow (e^y = T) \tag{3c}$$

$$(l^y_* = -1) \vee \eta_y \ \rightarrow (l^y = -1) \tag{3d}$$

In the example in Figure 4, $\eta_y = 1$ makes false conditions in equations (3a) and (3b) and true equations (3c) and (3d), obtaining that $e^y = T$ (the initial value of y is maintained until time T) and $l^y = -1$ (the final value of y is available since time -1), meaning that signal y never changes.

A more complete example is shown in Figure 5, where the waveforms are computed using the described method for the circuit in Figure 1.

The constraints describing other kinds of gates (NOR, AND, OR, and NOT) can be computed in a similar way. In non-monotonic gates such as XOR and XNOR, the early arrival time e_* of the output is equal to the minimum arrival time of the inputs, and the latest stabilization time l_* is equal to the maximum stabilization time of the inputs.

3 Preliminaries

ASP is a form of declarative programming oriented toward difficult combinatorial search problems (Erdem *et al.* 2016; Son *et al.* 2023; Bertagnon and Gavanelli 2024b). It relies on the stable model semantics, also known as answer set semantics (Gelfond and Lifschitz 1988). An ASP program Π consists of a finite set of rules, each of which is an implication of the form $H{:-}B$, where H is the head and B is the body of the rule.

The head H can be an atom, a choice atom, or an aggregate atom. An atom has the form $\mathtt{a}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$, where $\mathtt{t}_1, \ldots, \mathtt{t}_n$ are terms. A choice atom has the form

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

$\{\mathtt{a}(\mathtt{t}_1,\ldots,\mathtt{t}_n)\}$, and an aggregate atom can be of the form $A\{\mathtt{t}_1,\ldots,\mathtt{t}_m:\mathtt{c}_1,\ldots,\mathtt{c}_m\}\circ\mathtt{n}$, where A can be `#sum`, `#min`, `#max`, or `#count`, $\circ$ is a relational operator and `n` is an integer.

The body B is a set of literals, which can be either positive (`a`) or negative (`not a`), where `a` can be an atom, an aggregate or a condition. Literals and rules containing no variables are called ground. The ground instantiation of a program Π, denoted as $gr(\Pi)$, consists of all ground instances of rules in Π. A condition has the syntax $\mathtt{a}(\mathtt{X}):\mathtt{c}(\mathtt{X})$, where $\mathtt{a}(\mathtt{X})$ is an atom and $\mathtt{c}(\mathtt{X})$ is a condition that must be satisfied. This allows for the instantiation of variables to collections of terms within a single rule. For example, the rule $\mathtt{q}:-\mathtt{r}(\mathtt{X}):\mathtt{p}(\mathtt{X})$ is expanded to a conjunction of $\mathtt{r}(\mathtt{X})$ for all `X` that satisfy $\mathtt{p}(\mathtt{X})$.

Rules with an empty body are called facts, while rules with an empty head are called Integrity Constraints (ICs). The head of an IC is intended to be false.

An interpretation I of a program Π is a subset of the set of atoms occurring in Π. Atoms in I are considered true, while all remaining atoms are false. The reduct Π^I of a program Π with respect to an interpretation I is obtained by removing rules containing negative literals `not a`, where $\mathtt{a}\in I$, and removing all negative literals from the remaining rules. An interpretation I is a stable model of Π if it is a minimal model of Π^I.

ASP solvers typically work in two stages: grounding and solving. In the grounding stage, the program is converted into an equivalent ground program. The solving stage involves finding stable models (answer sets) of the ground program.

4 Problem formalization in answer set programming

A combinational circuit is encoded in ASP using two types of facts. Each logic gate is represented by a fact `gate_delay(Out, Gate, Delay)`, where `Gate` is the type of the gate, which can be `and`, `nand`, `or`, `nor`, `xor`, `xnor`, `inv` (inverter)[1] or `buff` (buffer); `Out` is the output signal of the gate, and is also used as a unique identifier of the gate, since a signal can be output only of one gate; `Delay` is the delay associated with the gate.

In order to represent uniformly gates with varying number of input signals, a fact `gate_in(Out, Gate, Input)` specifies that the gate of type `Gate`, whose output is `Out`, receives `Input` as one of the input signals. Each input is described through a separate fact, allowing gates to have an arbitrary number of inputs when applicable. For instance, a 3-input NAND gate with inputs `a`, `b`, and `c` and output `y` is represented by the following three facts: `gate_in(y, nand, a). gate_in(y, nand, b). gate_in(y, nand, c).`

To reason about signal behavior in the circuit, we define terms representing signals (Figure 6). A signal is any node appearing as an input or output of at least one gate (lines 1–2). From this, we automatically identify PIs and POs. A signal is a PI if it is never found as a gate output (line 4), and as a PO if it is never used as a gate input (line 3). With these definitions, the inputs and outputs are automatically inferred from the gate-level structure, with no need for external annotations or manual specification; modern grounders can nevertheless convert these definitions into a set of facts, which are handled very efficiently.

1 We did not use `not` because it is a reserved word in ASP.

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

```
1  signal(V) :- gate_in(V, _, _).
2  signal(V) :- gate_in(_, _, V).
3  output_node(V) :- signal(V), not gate_in(_, _, V).
4  input_node(V) :- signal(V), not gate_in(V, _, _).
5  boolean(0). boolean(1).
6  input_vec_no(1). input_vec_no(2).
7  1 = {v(V, InpVec, X) : boolean(X)} :- input_node(V), input_vec_no(InpVec).
8  v(Y, InpVec, V) :- gate_in(Y, buff, A), v(A, InpVec, V).                          % Buffer gate
9  v(Y, InpVec, Vout) :- gate_in(Y, inv, A), v(A, InpVec, Vin), inv(Vin,Vout).      % NOT gate (inverter)
10 inv(0, 1). inv(1, 0).
11 non_controlling((and;nand), 1). controlling((and;nand), 0).
12 non_controlling((or;nor), 0). controlling((or;nor), 1).
13 out_val(and, non_controlling, 1). out_val(and, controlling, 0).
14 out_val(or, non_controlling, 0). out_val(or, controlling, 1).
15 out_val(nand,Element,1-OutVal):- out_val(and,Element,OutVal).
16 out_val(nor, Element,1-OutVal):- out_val(or, Element,OutVal).
17 v(Y, InpVec, OutValue) :-                    % AND, OR, NAND, NOR with a controlling value in input
18     gate_in(Y, Gate, In),
19     controlling(Gate, Controlling),
20     out_val(Gate,controlling,OutValue),
21     v(In, InpVec, Controlling).
22 v(Y, InpVec, OutValue) :-            % AND, OR, NAND, NOR when all inputs are non-controlling values
23     gate_in(Y, Gate, _), input_vec_no(InpVec),
24     non_controlling(Gate, NonControlling),
25     out_val(Gate, non_controlling, OutValue),
26     v(In, InpVec, NonControlling) : gate_in(Y, Gate, In).
27 v(Y, InpVec, OutValue) :-                                              % xor, xnor with same input
28     gate_in(Y, Gate, A), gate_in(Y, Gate, B), A != B,
29     out_on_equal(Gate, OutValue),
30     v(A, InpVec, VA), v(B, InpVec, VB), VA == VB.
31 v(Y, InpVec, InvOutValue) :-                                        % xor, xnor with different input
32     gate_in(Y, Gate, A), gate_in(Y, Gate, B), A != B,
33     out_on_equal(Gate, OutValue), inv(OutValue, InvOutValue),
34     v(A, InpVec, VA), v(B, InpVec, VB), VA != VB.
35 out_on_equal(xor, 0). out_on_equal(xnor, 1).
```

Fig. 6. Encoding in ASP of signals and logic gates, common to all the encodings.

We consider the behavior of a combinational circuit under a pair of input vectors as introduced in Section 2. We define a domain of Boolean values, in line 5, and specify the two input vectors in line 6.

As the objective is to find two input vectors that witness the required properties (e.g., longest delay), we introduce in line 7 a choice rule that selects exactly one Boolean value for each input vector and each PI. The atom `v(V, InpVec, X)` means that signal `V` takes the Boolean value `X` under input vector `InpVec`.

For non-PI signals, the logical value is determined by the functional behavior of the gates driving them; again, each signal takes a boolean value for each input vector. In the case of unary gates, such as `buff` (buffer) and `inv` (inverter), the output value is determined by the value of their single input signal (lines 8–10). These rules are applied for each input vector (`InpVec`$\in 1, 2$).

In order to define compactly the logic behavior of multi-input gates, avoiding long lists of cases, we leverage on the concept of *controlling* and *non-controlling* values (Def. 1). Value 0 is controlling for AND and NAND, while 1 is their non-controlling element (line 11). Value 1 is controlling for OR and NOR, while 0 is non-controlling (line 12). Lines 13–16 provide the truth table of these four gate types relying only on the controlling and non-controlling values.

The output signal values for gates belonging to types enjoying controlling values are computed in lines 17–26. The first clause (lines 17–21) considers the case in which at least one input takes the controlling value; the second (lines 22–26) when all inputs

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

```
36 t(V, InpVec, 0) :- input_node(V), input_vec_no(InpVec).
37 eta(SignalY) :- fixed(SignalY),                                  % Definition of eta, see equation (2)
38     ts(SignalY, 1, Ey), ts(SignalY, 2, Ly), Ly-Ey <= 0.
39 fixed(SignalY) :- v(SignalY, 1, Vy), v(SignalY, 2, Vy).
40 t(SignalY, 1, EyStar+Delay) :- EyStar != MaxTime, maxtime(MaxTime),                    % Eq (3a)
41     not eta(SignalY), ts(SignalY, 1, EyStar), gate_delay(SignalY, _, Delay).
42 t(SignalY, 2, LyStar+Delay) :- LyStar != -1,                                          % Eq (3b)
43     not eta(SignalY), ts(SignalY, 2, LyStar), gate_delay(SignalY, _, Delay).
44 t(SignalY, 1, MaxTime) :- maxtime(MaxTime), ts(SignalY, 1, MaxTime).     % First part of Eq (3c)
45 t(SignalY, 1, MaxTime) :- maxtime(MaxTime), eta(SignalY).               % Second part of Eq (3c)
46 t(SignalY, 2, -1) :- ts(SignalY, 2, -1).                                % First part of Eq (3d)
47 t(SignalY, 2, -1) :- eta(SignalY).                                     % Second part of Eq (3d)
48 static_timing_analysis(V, 0) :- input_node(V).
49 static_timing_analysis(Out, MaxIn + Delay) :- gate_in(Out, Gate, In),
50        gate_delay(Out, Gate, Delay), static_timing_analysis(In,MaxIn).
51 maxtime(MaxTime+1) :- #max{T: static_timing_analysis(V, T), output_node(V)} = MaxTime.
52 max_output_delay(MaxOutDelay) :- MaxOutDelay = #max{L, OutNode : t(OutNode, 2, L), output_node(OutNode)}.
53 #maximize{ MaxOutDelay: max_output_delay(MaxOutDelay) }.
```

Fig. 7. Time computation and optimization.

take the non-controlling value. The conditional atom `v(In, InpVec, NonControlling) : gate_in(Y, Gate, In)` is expanded at grounding time into a conjunction of atoms `v(In, InpVec, NonControlling)`, one for each ground atom satisfying `gate_in(Y, Gate, In)`; in the example of a NAND gate with inputs `a`, `b` and `c`, it is expanded to the conjunction `v(a, InpVec, NonControlling)`, `v(b, InpVec, NonControlling)`, `v(c, InpVec, NonControlling)`, so it is true only of all the three inputs take the non-controlling value.

Finally, to simplify the exposition, we show the code for modeling XOR and XNOR gates having exactly two inputs. The code can be extended also for multi-inputs but is not necessary for common gate libraries used in benchmarks, where XOR/XNOR gates are generally with two inputs. The auxiliary predicate `out_on_equal/2` (line 35) defines the output value of these gate types when the two inputs are equal. A XOR gate produces 0 when both inputs are equal, whereas XNOR produces 1. The logical behavior is encoded through two rules. The first handles the case where the inputs have the same value (lines 27–30), and the gate produces the output defined by `out_on_equal/2`. The second rule, lines 31–34, covers the case with different input values.

Figure 7 shows the code for computing the early ($e_{\mathbf{v}}$) and late ($l_{\mathbf{v}}$) arrival times of each signal according to the timing model introduced in Section 2. Table 1 summarizes how the variables in equations (1)–(3) are encoded in ASP. Since early and late arrival are times associated to the first and second bit vector, both are represented in ASP through the same predicate `t(Signal, InpVec, Time)`, where `InpVec = 1` refers to the early arrival time and `InpVec = 2` to the latest stabilization time. For PI nodes, these times are fixed to zero, as shown in line 36.

The auxiliary variables e_* and l_* are represented by the predicate `ts(Signal, InpVec, Time)`; `ts` stands for *Time Star*, and the parameters have the same meaning as in `t`/3.

Lines 37–38 define η as in equation (2). Equations (3a) and (3b) are implemented in lines 40–43, where the arrival times are computed by adding the gate delay to the early (e_*^y) and late (l_*^y) auxiliary values. Equation (3c) is implemented in lines 44–45 while equation (3d) corresponds to clauses 46–47.

Equations (3a) and (3c) require a value T that is higher than the maximum delay; while for correctness any value sufficiently large is valid, larger values can increase the solving time. For this reason, we compute, in lines 48–50, the upper bound with STA.

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

Table 1. *Symbols in the mathematical formulation (Section 2) and in the ASP code*

Symbol	ASP	Description
e^v	`Ev` in t(v,1,Ev)	Early Arrival Time of signal v
l^v	`Lv` in t(v,2,Lv)	Late Arrival Time of signal v
e^v_*	`Ev` in ts(v,1,Ev)	Auxiliary Early Arrival Time of signal v
l^v_*	`Lv` in ts(v,2,Lv)	Auxiliary Late Arrival Time of signal v
η_v	`eta(v)`	Masked potential hazard in signal v
T	`maxtime(T)`	A time larger than the possible maximum delay

Probably, the most intuitive formulation would be to state that, for each gate G, the maximum possible output delay is the maximum of the inputs with added the delay of the gate:

```
static_timing_analysis(V, 0) :- input_node(V).

static_timing_analysis(Out, MaxIn + Delay) :- gate_delay(Out, _, Delay),
    MaxIn = #max { Time : gate_in(Out, _, In), static_timing_analysis(In, Time) }.
```

This version is correct, but contains a recursion through the `#max` aggregate; this recursion hinders the optimization of the gringo grounder, that is unable to compute the upper bound at grounding time, leaving it to be computed at solving time. The formulation in lines 48–50, instead, is stratified, and the maximum time is computed directly by the gringo grounder. The size of the ground program is also significantly reduced. In the formulation in lines 48–50, it can be the case that for some gate, the value of the maximum time is not unique; in order to obtain the global maximum, it is enough to apply the `#max` aggregate, as shown in line 51.

Finally, the objective function is to maximize the maximum delay (lines 52–53).

The rest of the code is devoted to computing the value of the auxiliary variables e^y_* and l^y_* for each of the gates types, based on equations (1a)–(1h); in the next sections, we first show an intuitive encoding, then an advanced encoding for this task.

4.1 A first encoding

In the first encoding (Figure 8), equations (1a)–(1h) are generalized to the case of multiple inputs; in order to reduce the number of cases, we rely again on the concept of controlling value, and the eight equations (1a)–(1h) are rewritten as four clauses. The clauses on lines 57 and 67 compute, respectively, the e^y_* and l^y_* arrival times in case there is no controlling value in input to the gate; from equation (1a), $e^y_* = \min\{e^i\}$ where i is an input to the gate, while (equation 1e) $l^y_* = \min\{l^i\}$. Clauses 62 and 72, instead, consider the case in which at least one of the inputs is equal to the controlling value. In such a case, e^y_* is the maximum of the e^i, where i are the inputs taking the controlling value

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

```
57 ts(SignalY, 1, MinE) :-                % Early* Arrival time in AND, OR, NAND, NOR gates when there is
58     gate_in(SignalY, Gate, _),                                % no controlling value in input, Eq (1a)
59     controlling(Gate, Controlling),
60     #count{ I : gate_in(SignalY, Gate, I), v(I, 1, Controlling) } = 0,
61     MinE = #min { T : gate_in(SignalY, Gate, I), t(I, 1, T) }.
62 ts(SignalY, 1, MaxE) :-                % Early* Arrival time in AND, OR, NAND, NOR gates when there is
63     gate_in(SignalY, Gate, _),             % at least one controlling value in input, Eq (1b),(1c),(1d)
64     controlling(Gate, Controlling),
65     #count{ I : gate_in(SignalY, Gate, I), v(I, 1, Controlling) } > 0,
66     MaxE = #max { T : gate_in(SignalY, Gate, I), v(I, 1, Controlling), t(I, 1, T) }.
67 ts(SignalY, 2, MaxL) :-                 % Late* Arrival time in AND, OR, NAND, NOR gates when there is
68     gate_in(SignalY, Gate, _),                                % no controlling value in input, Eq (1e)
69     controlling(Gate, Controlling),
70     #count{ I : gate_in(SignalY, Gate, I), v(I, 2, Controlling) } = 0,
71     MaxL = #max { T : gate_in(SignalY, Gate, I), t(I, 2, T) }.
72 ts(SignalY, 2, MinL) :-                 % Late* Arrival time in AND, OR, NAND, NOR gates when there is
73     gate_in(SignalY, Gate, _),             % at least one controlling value in input, Eq (1f),(1g),(1h)
74     controlling(Gate, Controlling),
75     #count{ I : gate_in(SignalY, Gate, I), v(I, 2, Controlling) } > 0,
76     MinL = #min { T : gate_in(SignalY, Gate, I), v(I, 2, Controlling), t(I, 2, T) }.
77 ts(SignalY, 1, EA) :- SignalA != SignalB, EB >= EA,           % Early* Arrival time in XOR, XNOR gates
78     t(SignalA, 1, EA), gate_in(SignalY, (xor;xnor), SignalA),
79     t(SignalB, 1, EB), gate_in(SignalY, (xor;xnor), SignalB).
80 ts(SignalY, 2, LB) :- SignalA != SignalB, LB >= LA,            % Late* Arrival time in XOR, XNOR gates
81     t(SignalA, 2, LA), gate_in(SignalY, (xor;xnor), SignalA),
82     t(SignalB, 2, LB), gate_in(SignalY, (xor;xnor), SignalB).
83 ts(SignalY,InpVec,TA):-                              % Early* and Late* Arrival time in unary gates
84     t(SignalA,InpVec,TA), gate_in(SignalY,(inv;buff),SignalA).
```

Fig. 8. Basic encoding.

(equations (1b), (1c), (1d)), while l^y_* is the minimum of the l^i that are equal to the controlling value (equations (1f), (1g), (1h)).

Considering XOR and XNOR gates, clause 77 computes the e^y_*, while clause 80 computes the l^y_*. Finally, clause 83 takes care of unary gates (inverter and buffer).

4.2 An advanced encoding

The encoding proposed in Section 4.1 implements correctly the equations (1a)–(1h), but it suffers from the fact that the number of clauses in the ground program can be very large. Consider, for example, the clause in line 57 (corresponding to equation (1a)); to simplify the exposition, let us consider a simplified version that holds only for a NAND gate with two inputs, only for the early arrival time (so we remove the parameter `InpVec` from all predicates to simplify the exposition):

```
ts(Y, EA) :- v(A,1), v(B,1), A!=B, % Both inputs are 1
    gate_in(Y, Gate, A), t(A, EA), % Compute EA = time of input A
    gate_in(Y, Gate, B), t(B, EB), % Compute EA = time of input B
    EA <= EB.
```

Such a clause intuitively means that if both inputs are 1, then the early arrival time $e^{\mathtt{Y}}_*$ is the minimum of the two inputs $e^{\mathtt{A}}$ and $e^{\mathtt{B}}$. If `EA` and for `EB` can possibly take t values each, such a clause is grounded into $O(t^2)$ clauses. For a gate with k inputs, the number of clauses worsens to $O(t^k)$.

However, one can observe that in order to compute the minimum, it is not strictly necessary to know the exact timing of *all* the input signals: it is necessary only to know the exact timing of the smallest one, plus it is needed to have a proof that all the other signals have a larger (or equal) time. So, if we define a predicate `tgeq(S, Time)` that

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

```
94  tgeq(S, InpVec, V) :- t(S, InpVec, V).
95  tgeq(S, InpVec, V-1) :- V>=0, tgeq(S, InpVec, V).
96  ts(SignalY, 1, EA) :- t(SignalA, 1, EA),
97      all_non_controlling_except(SignalY,1,SignalA),
98      tgeq(SignalB, 1, EA) : gate_in(SignalY, Gate, SignalB), SignalB != SignalA.      % EA is the minimum
99  ts(SignalY, 1, EB) :- t(SignalB, 1, EB),
100     gate_in(SignalY, Gate, SignalB), v(SignalB, 1, V), controlling(Gate, V),
101     not tgeq(SignalA, 1, EB + 1) : gate_in(SignalY, Gate, SignalA), v(SignalA, 1, V), SignalA != SignalB.
102 ts(SignalY, 2, LA) :- t(SignalA, 2, LA),
103     all_non_controlling_except(SignalY,2,SignalA),
104     not tgeq(SignalB, 2, LA + 1) : gate_in(SignalY, Gate, SignalB), SignalB != SignalA.
105 ts(SignalY, 2, LB) :- t(SignalB, 2, LB),
106     gate_in(SignalY, Gate, SignalB), v(SignalB, 2, V), controlling(Gate, V),
107     tgeq(SignalA, 2, LB) : gate_in(SignalY, Gate, SignalA), v(SignalA, 2, V), SignalA != SignalB.
108 ts(SignalY, 1, Min) :- t(SignalA, 1, Min),
109     gate_in(SignalY, (xor;xnor), SignalA),
110     gate_in(SignalY, (xor;xnor), SignalB),
111     SignalA != SignalB, tgeq(SignalB, 1, Min).
112 ts(SignalY, 2, Max) :- t(SignalA, 2, Max),
113     gate_in(SignalY, (xor;xnor), SignalA),
114     gate_in(SignalY, (xor;xnor), SignalB),
115     SignalA != SignalB, not tgeq(SignalB, 2, Max + 1).
116 all_non_controlling_except(SignalY,BitVec,SignalA):- % all inputs to the gate having output SignalY take
117     input_vec_no(BitVec),                           % the non-controlling value, except possibly SignalA
118     gate_in(SignalY, Gate, SignalA), non_controlling(Gate, V),
119     v(SignalB, BitVec, V) : gate_in(SignalY, Gate, SignalB), SignalB != SignalA.
```

Fig. 9. Advanced encoding.

is true when signal `S` has a time greater than or equal to[2] `Time`, we would be able to compute the minimum using only a linear number of clauses with respect to t, as follows:

```
ts(Y, EA) :- v(A,1), v(B,1), A!=B,
    gate_in(Y, Gate, A), t(A, EA), % Compute EA = time of input A
    tgeq(B,EA). % Ensure that EB >= EA
```

Note that in this version, variable `EB` does not occur in the clause, as it is not necessary to know the exact value of $e^{\mathtt{B}}$: it is only necessary to know that $e^{\mathtt{B}}$'s value is higher than or equal to $e^{\mathtt{A}}$, which is stated by the atom `tgeq(B,EA)`. Now, predicate `tgeq(S,T)`, with the intuitive meaning that $e^{\mathtt{S}} \geq \mathtt{T}$, can be defined as:

```
tgeq(S, T) :- t(S, T).
tgeq(S, T-1) :- T>=0, tgeq(S, T).
```

The first clause states that if $e^{\mathtt{S}} = \mathtt{T}$, then obviously $e^{\mathtt{S}} \geq \mathtt{T}$, while the second says that if $e^{\mathtt{S}} \geq \mathtt{T}$, then also $e^{\mathtt{S}} \geq \mathtt{T} - 1$. Note that predicate `tgeq` is defined with two Horn clauses (which are handled very efficiently by modern ASP solver – it is worth to remember that Horn-SAT is a polynomially solvable fragment of SAT), and that as soon as the ASP solver infers that atom `t(S,T0)` is true for some specific value `T0`, the truth of `t(S,T)` is inferred $\forall \mathtt{T} \leq \mathtt{T0}$ through the simple unit propagation mechanism, in linear time.

Figure 9 shows the computation of e_* and l_* in the advanced encoding; the complete code of the advanced encoding consists of the code in Figures 6, 7, and 9.

The clause in lines 99–101 considers the case in which there is at least one input taking the controlling value in the first bit vector: in such a case, the early arrival time e_*^Y of the output is the maximum among the early arrival times of the inputs at the controlling value. This is implemented with a clause stating that the early arrival $e_*^Y = e^B$ for some

2 The acronym `tgeq` stands for `t`ime is `g`reater than or `eq`ual to.

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

signal B taking the controlling value and with early arrival time e^B that is higher than or equal to all other inputs having the controlling value.

Another improvement in this encoding can be found in the clause in lines 96–98. Consider equation (1a): in the case of multiple inputs it could be interpreted as: if all the inputs take the non-controlling value, then the time e_*^Y is the minimum of the early arrival time of the inputs. This could be implemented as

```
ts(Y, 1, EA) :-
    t(A, 1, EA), v(A, 1, NonControlling), non_controlling(Gate, NonControlling),
    v(B, 1, NonControlling) : gate_in(Y, Gate, B), B != A;
    tgeq(B, 1, EA) : gate_in(Y, Gate, B), B != A;
    gate_in(Y, Gate, A).
```

That is, if the input `SignalA` has early arrival time lower than or equal to all other inputs, and all the inputs take the non-controlling value in the first bit vector, then $e_*^{\mathtt{Y}} = e^{\mathtt{A}}$. On the other hand, if there exists an input `A` having early arrival time lower than or equal to all other inputs but taking the controlling value (while all other inputs take the non-controlling value), then again $e_*^{\mathtt{Y}} = e^{\mathtt{A}}$, this time due to equation (1d). So, it is possible to strengthen the clause, removing the condition `v(A, 1, NonControlling)`, as in lines 96–98. With the strengthened clause, the ground program is smaller, and unit propagation can be applied more often, since the clause is shorter.

5 Experimental results

In order to evaluate the running time of our method, we performed experiments on circuits from the ISCAS85 (Brglez and Fujiwara 1985) and ITC99 (Corno *et al.* Corno et al., 2000) benchmark suites.[3] The output signal of a gate G can be connected in input to a number of other gates; such a number is called *fan-out*, and it influences the delay associated to the gate G. In the experiments, we used for each gate a delay proportional to the fan-out of the gate; of course, the same methodology and encoding could be used with delays computed with other methodologies (e.g., hardware simulations).

We compared the basic encoding presented in Section 4.1 and the advanced encoding in Section 4.2. Table 2 reports the number of logic gates for each circuit, as well as the solving and grounding times, and the grounding size in MB. The results were obtained using clingo 5.7.1 (Gebser *et al.* 2019) on an AMD EPYC 9454 processor at 2.75 GHz, with a maximum of 64 GB of reserved memory. In the tests, we used clingo with only one core (without parallelism) and imposed a timeout (T/O) of 10 h (36,000 s). As the objective is to compare the two ASP approaches, we do not report the instances in which both methods incurred into a timeout.

The second encoding provides speedups from 1.5× to 18×. Instance C6288 is notoriously hard in the literature (Qiu and Walker 2003); it is a 16 bits × 16 bits multiplier that has an exponential number of paths, and many methods in the literature are not able to solve it in reasonable time. While the first encoding is unable to solve it within 10 hours, the advanced encoding can solve it in less than 4 hours. A similar behavior

3 In the ITC99 case, we considered the combinational fraction of these circuits.

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

Table 2. *Benchmark results for a set of logic circuits using the two ASP encoding. For each circuit, the number of logic gates, solving and grounding times, and grounding sizes are reported. The final two columns show the maximum observed late output and the maximum delay from static timing analysis*

		Basic encoding			Advanced encoding				
Name	# Logic gates	Total time [s]	Grounding time [s]	Grounding size [MB]	Total time [s]	Grounding time [s]	Grounding size [MB]	Max late output	Max STA delay
C432	152	1.21	0.33	51.4	0.71	0.29	7.8	71	71
C499	320	0.35	0.08	9.1	0.23	0.20	3.7	40	40
C880	311	7.98	1.46	25.4	2.21	1.11	5.9	72	72
C1355	601	28.95	4.55	94.9	18.10	2.71	20.6	76	76
C1908	274	297.32	13.93	17.6	57.49	8.00	5.0	106	118
C2670	761	56.07	5.27	41.0	18.13	9.13	11.6	108	112
C3540	996	1566.54	27.82	244.2	86.50	23.57	42.0	126	136
C5315	1605	331.32	23.33	145.1	104.00	37.49	38.1	134	138
C6288	1588	T/O	49.68	1582.5	13291.97	240.52	251.0	382	386
C7552	3510	1272.73	43.73	611.3	160.14	88.65	133.8	126	130
B11	622	21.78	2.72	54.8	5.09	1.71	16.4	92	105
B12	963	25.41	6.99	75.3	7.92	4.70	25.9	70	90
B13	310	0.43	0.36	2.5	0.32	0.28	2.1	35	35
B14	8567	T/O	1183.08	12729.1	12006.65	589.69	1858.1	256	259

is observed, for instance, B14, a subset of the Viper processor, but with a much larger grounding size that exceeds 12 GB for the basic encoding while remaining below 2 GB with the advanced encoding. This is likely due to the higher number of logic gates (8567 in B14 *vs.* 1588 in C6288), which significantly increases the grounding complexity.

6 Related work

The work most closely related to the current article is by Andres *et al.* (2013): they address a closely related problem in hardware design, namely the computation of the longest path in a combinational circuit, in ASP. Their approach is based on choosing a gate in the circuit, which is the gate under test, and restricting their attention only on the paths that pass through that gate. This can help avoiding, in some instances, large parts of the circuit, resulting in a very quick search. Afterward, they find the longest path (varying the gate under test) such that all the gates in the path commute when the input array switches from the first array to the second. In order to increase further the speed of the computation, they adopt a multi-shot solving strategy (Gebser *et al.* 2019). Indeed, their solution is extremely quick in instances having a number of paths not exceedingly high. Our approach is, in most instances, slower, but more precise, because it solves a harder problem, taking into consideration also the hazards.

ASP was employed in a number of different applications in hardware design and verification. Ishebabi *et al.* (2009) address a problem of automated design for multiprocessor systems on FPGAs. Andres *et al.* (2013) propose in ASP an automated system design approach for embedded computing systems.

Gavanelli *et al.* (2017) address a problem of designing an optical router on-chip maximizing parallelism while avoiding routing faults. They provide and compare approaches based on ASP, Constraint Logic Programming, and Integer Linear Programming.

Bobda *et al.* (2018) tackle system-level synthesis for heterogeneous multi-processors on a chip using ASP. ASP is also at the basis of a stream reasoning tool that was used for monitoring and scheduling in a semiconductor failure analysis lab (Mastria *et al.* 2024).

Digital hardware verification is a complex process spanning from logic synthesis to physical design. For synchronous systems, functional and timing verification can be decoupled; this work focuses on the latter, to ensure that no timing violations occur.

Initial efforts centered on STA (Hitchcock 1982; Agrawal 1982; Chadha and Bhasker 2009), later evolving to address false path identification (Kundu 1994). Early methods employed multi-valued algebras and custom justification algorithms to handle hazards. More recent approaches leverage SAT solvers for signal justification (Eggersglüss *et al.* 2010) and target critical path identification in complex modules (Pomeranz 2024).

Alternative strategies include event-driven simulation to exploit higher clock frequencies while mitigating false paths (Wang and Robinson 2019), and machine learning techniques like association rule analysis to identify input vectors critical to reliability (Shi *et al.* 2024). However, these methods offer approximate rather than exact solutions.

7 Conclusions

We proposed an ASP-based solution for computing the maximum delay in combinational circuits, a key task in hardware design and verification. This work, a collaboration

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

between logic programming and hardware design experts, adopts a fine-grained approach that considers hazards, unlike the mainstream method of finding the maximum sensitizable path. Experiments show our approach can solve instances considered very hard in the literature. Future work includes experimenting with ASP modulo difference logic (Janhunen *et al.* 2017).

Competing interests

The authors declare none.

References

Agrawal, V. 1982. Synchronous path analysis in MOS circuit simulator. In *19th Design Automation Conference*, 629–635.

Andres, B., Gebser, M., Schaub, T., Haubelt, C., Reimann, F. and Glass, M. 2013a. Symbolic system synthesis using answer set programming. In *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, P. Cabalar and T. C. Son, Eds. Springer, 8148, 79–91.

Andres, B., Sauer, M., Gebser, M., Schubert, T., Becker, B. and Schaub, T. 2013b. Accurate computation of sensitizable paths using answer set programming. In *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, P. Cabalar and T. C. Son, Eds. Springer, vol. 8148, 92–101.

Bertagnon, A. and Gavanelli, M. 2024a. ASPECT: Answer Set rePresentation as vEctor graphiCs in laTex. *Journal of Logic and Computation* 34, 8, 1580–1607.

Bertagnon, A. and Gavanelli, M. 2024b. Geometric reasoning on the euclidean traveling salesperson problem in answer set programming. *Intelligenza Artificiale* 18, 1, 139–152.

Bobda, C., Yonga, F., Gebser, M., Ishebabi, H. and Schaub, T. 2018. High-level synthesis of on-chip multiprocessor architectures based on answer set programming. *Journal of Parallel and Distributed Computing* 117, 161–179.

Brglez, F. and Fujiwara, H. 1985. A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN. In *Proc. of the 1985 IEEE Int. Symposium on Circuits and Systems (ISCAS'85)*.

Chadha, R. and Bhasker, J. 2009. *Static Timing Analysis for Nanometer Designs. A Practical Approach*. Springer, New York, NY, USA.

Corno, F., Reorda, M. S. and Squillero, G. 2000. *RT-Level ITC 99 Benchmarks and First ATPG Results*. IEEE Design & Test of Computers, Los Alamitos, CA, USA, 44–53.

Eggersglüss, S., Fey, G., Glowatz, A., Hapke, F., Schloeffel, J., Drechsler, R. 2010. MONSOON : a SAT-based ATPG for path delay faults using multiple valued logics. *Journal of Electronic Testing: Theory and Application* 26, 307–321.

Erdem, E., Gelfond, M. and Leone, N. 2016. Applications of answer set programming. *AI Magazine* 37, 3, 53–68.

Gavanelli, M., Nonato, M., Peano, A. and Bertozzi, D. 2017. Logic programming approaches for routing fault-free and maximally parallel wavelength-routed optical networks-on-chip (application paper). *Theory and Practice of Logic Programming 17*, 800–818.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming 19*, 1, 27–82.

https://doi.org/10.1017/S1471068425100288 Published online by Cambridge University Press

Gelfond, M. and Lifschitz, V. (1988) The stable model semantics for logic programming. In *International Conference on Logic Programming*, R. A. Kowalski and K. A. Bowen, Eds.

Gharaybeh, M., Agrawal, V. and Bushnell, M. 1998. False-path removal using delay fault simulation. In *Proc. 7th Asian Test Symposium (ATS'98) (Cat. No.98TB100259)*, 82–87.

Hitchcock, R. 1982. Timing verification and the timing analysis program. In *19th Design Automation Conference*, 594–604.

Ishebabi, H., Mahr, P., Bobda, C., Gebser, M. and Schaub, T. 2009. Answer set versus integer linear programming for automatic synthesis of multiprocessor systems from real-time parallel programs. *International Journal of Reconfigurable Computing 2009*, 863630:1–863630:11.

Janhunen, T., Kaminski, R., Ostrowski, M., Schellhorn, S., Wanko, P. and Schaub, T. 2017. Clingo goes linear constraints over reals and integers. *Theory and Practice of Logic Programming 17*, 872–888.

Kundu, S. 1994. An incremental algorithm for identification of longest (shortest) paths. *Integration-the Vlsi Journal 17*, 1, 25–31.

Lu, X., Li, Z., Qiu, W., Walker, D. M. H. and Shi, W. 2004. Longest path selection for delay test under process variation. In *Proc. ASP-DAC*. IEEE Press, 98–103.

Mastria, E., Pagliaro, D., Calimeri, F., Perri, S., Pleschberger, M. and Schekotihin, K. (2024) Monitoring and scheduling of semiconductor failure analysis labs. In *LPNMR*, vol. 15245, C. Dodaro, G. Gupta and M. V. Martinez, Eds. Springer.

Pomeranz, I. 2024. Longest path selection based on path identifiers. *IEEE Access* 12, 14512–14520.

Qiu, W. and Walker, D. 2003. Testing the path delay faults of ISCAS85 circuit c6288. In *Proc. 4th Int. Workshop on Microprocessor Test and Verification*, 19–24.

Shi, Z., Xiao, J., Jiang, J., Zhang, Y. and Zhou, Y. 2024. ARA-RCIV: identifying reliability-critical input vectors of logic circuits based on the association rules analysis approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 43*, 8, 2479–2492.

Son, T. C., Pontelli, E., Balduccini, M. and Schaub, T. 2023. Answer set planning: a survey. *Theory and Practice of Logic Programming 23*, 1, 226–298.

Wang, X. and Robinson, W. H. 2019. Error estimation and error reduction with input-vector profiling for timing speculation in digital circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 38*, 2, 385–389.