

An Efficient Algorithm to Verify Generalized False Paths

Olivier Coudert
OC Consulting
Auf der Höh 16a
83607 Holzkirchen, Germany
Skype: ocoudert Twitter: @ocoudert
olivier_coudert@ocoudert.com

ABSTRACT

Timing exception verification has become a center of interest as incorrect constraints can lead to chip failures. Proving that a false path is valid or not is a difficult problem because of the inherent computational cost, and because in practice false paths are not specified one full path at a time. Instead designers use *generalized* false paths, which represent a set of paths. For instance the SDC format (Synopsys Design Constraint) specifies false path exceptions using a “–from –through –to” syntax that applies on sets of pins, often using wildcards to denote these sets. This represents many (usually hundreds to thousands) actual full paths. This paper proposes a method to verify generalized false paths in a very efficient manner. It is shown to be about 10x faster than the current state-of-the-art, making false path verification an overnight task or less for multi-million gate designs.

Categories and Subject Descriptors

B.6.3 [Logic Design]: Design Aids – Verification

General Terms

Verification, Algorithms, Performance, Reliability, Design.

Keywords

Formal verification, timing exception, false path, generalized false path, SDC, correctness, sensitization, co-sensitization, SAT.

1. INTRODUCTION

Timing exceptions, like false paths and multi-cycle paths, are used to remove some of the pessimism inherent to static timing analysis (STA) algorithms. As a side effect, timing exceptions allow logic synthesis to do a better job when optimizing the netlist. However an improper set of timing exceptions may lead to a chip failure. Thus timing exception verification is receiving more and more attention. Several companies propose timing exception verification products, e.g., Fishtail (Confirm), Real Intent (PureTime), Atrenta (SpyGlass-Constraints), Averant (SolidTC), and Cadence (Conformal).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC'10, June 13-18, 2010, Anaheim, California, USA

Copyright 2010 ACM 978-1-4503-0002-5 /10/06...\$10.00

Given an RTL or netlist description and a set of timing exceptions (usually expressed with a SDC file), the problem consists of verifying the correctness of the timing constraints on the design. There are a number of models to define the semantics of a timing exception and a number of ways to check its correctness. Among the several models, one can distinguish between the delay-dependent models and the delay-independent models.

A delay-dependent model accounts for the actual delay in the signal propagation to decide whether a path is true or not [2] [7]. A delay-independent model defines a true or false path so that it does not depend on any signal propagation delay. Therefore the later definition is valid regardless of the delay library. In practice, a delay-independent validation is required, because (1) the designer expects timing constraints to be useable with any library; and (2) the relevance of the delay-dependent verification depends on the accuracy of the delay model itself, which is still an approximation; thus a delay-dependent verification does not guarantee a fault-free circuit.

Complete and practical methods have been proposed to check multi-cycle paths [12]. However, false path verification remains a bottleneck from the computational point of view. A pure path exploration-based method cannot be used to process generalized false paths (e.g., written in SDC), because a single generalized false path captures a large number of actual paths (usually hundreds to thousands and more). A typical SDC deck is made of a several thousands generalized false path constraints, which makes industrial-scale false path verification quite challenging.

This paper therefore addresses the delay-independent verification of generalized false path. Section 2 first explains the common concepts of static sensitization and co-sensitization, and focuses on two criterions: (i) the path is not statically co-sensitizable; and (ii) the path is statically sensitizable. If (i) is true, the path is false. If (ii) is true, the path is true. Note that if a path does not satisfy (i) and (ii), its true/false nature depends on the delay assignment. It is incorrect to state that a path is false when (ii) fails, despite claims sometimes found in the industry.

The paper then shows that straightforward applications of criteria (i) and (ii) to a path-based formulation are not practical (Section 2.7). Instead, Section 3 introduces a recursive definition of the criterions (i) and (ii). This leads to two major improvements. First, there is no longer a path-per-path examination. Instead the subgraph induced by the generalized false path is processed in one single pass. Second, the recursive formulation of criterion (i) and (ii) produces a single SAT problem of size $O(n \ln n)$, where n is the size of the subgraph (Section 3.4). This algorithm dramatically

improves the performance of the generalized false path verification, to about one path per second, as shown in Section 4.

2. TIMING EXCEPTIONS

This section introduces some notations used in this paper.

2.1 Netlist

A netlist is made of logic gates and nets. Nets connect output pins to input pins. We assume the netlist to be cycle-free. The primary inputs (PI), and primary outputs (PO), of the netlist are the free Boolean variables, and outputs of the netlist, respectively.

Since a multi-output gate can be expanded into a forest of single-output gates, we assume without loss of generality that all gates have a single output. Given a gate g , we note x_1, x_2, \dots, x_k its input pins and y its output pin. We will often identify a gate g with its output y .

We refer the reader to [10] for an overview of static timing analysis (STA), which estimates the latest arrival times and earliest required times at each pin in a netlist.

2.2 Controlling and Controlled Values

An input pin has a controlling value if that value determines the output value of the gate independently of the values of the other inputs. E.g., 0 is a controlling value for a NAND gate. A controlled value is the value at the output of a gate if one of its inputs assumes a controlling value. E.g., 1 is a controlled value for a NAND gate. Note that a XOR gate has no controlling or controlled value.

Let $y(x_1, x_2, \dots, x_n)$ be a Boolean function. The Boolean difference of y w.r.t. variable x_k is defined as:

$$\frac{\partial y}{\partial x_k} = y(x_1, \dots, x_{k-1}, 0, x_{k+1}, \dots, x_n) \oplus y(x_1, \dots, x_{k-1}, 1, x_{k+1}, \dots, x_n)$$

Whenever the Boolean difference is 1, toggling the value of x_k results in toggling the value of y . If the Boolean difference is always 0, y does not depend on x_k .

2.3 Generalized Path and Timing Exception

Let N be a netlist, and P_1, P_2, \dots, P_n be sets of pins. A generalized path $G(P_1, \dots, P_n)$ is the set of all the paths from PI to PO that go through the ordered pins p_1, p_2, \dots, p_n , with $p_k \in P_k$ for $1 \leq k \leq n$. A pin that belongs to some P_k is called an anchor. Note that $G(\{y\})$, the set of paths going through the (unique) output pin y of gate g , is equal to $G(\{x_1, x_2, \dots, x_k\})$, the set of paths going through any input pin x_1, x_2, \dots, x_k of g . Therefore we can always restrict anchors to be input pins only. Also since we consider paths from PI to PO , we will assume without loss of generality that $P_1 \subseteq PI$ and $P_n \subseteq PO$.

Given a path, an on-input (or on-pin) is an input pin that belongs to the path. A side input (or side pin) is an input pin that does not belong to the path, but whose gate is on the path.

Following the SDC syntax [4], a timing exception is expressed as:

$$[-\text{from } \langle \text{pin} \rangle^*] [-\text{through } \langle \text{pin} \rangle^*] [-\text{to } \langle \text{pin} \rangle^*]$$

A missing “-from” is equivalent to “-from PI ”, and a missing “-to” is equivalent to “-to PO ”. A timing exception “-from P_1 -through P_2 -through P_3 ... -through P_{n-1} -to P_n ” is a generalized false path that notifies the timer that *all* paths in $G(P_1, \dots, P_n)$ are

false, i.e., they do not contribute to the actual arrival times (and required times). False paths are used to reduce the pessimism inherent to STA when estimating timing (STA is based on a MIN/MAX propagation, which does not account for logical dependencies between nets).

2.4 Sensitization

Let π be a path (y_1, y_2, \dots, y_n) from PI to PO , where gate y_{k-1} drives gate y_k . Path π is statically sensitizable [1] iff there exists an input vector such that:

$$\prod_{k=2}^{k=n} \frac{\partial y_k}{\partial y_{k-1}} \neq 0 \quad (1)$$

Equation (1) means that every gate y_k is sensitized by its driver y_{k-1} . If the Boolean variable y_1 (a primary input) is toggled, all the values along the path will toggle up to the primary output y_n .

For a netlist made of elementary gates (e.g., AND, OR, NOT, XOR), this is equivalent to the following:

Definition 1. A path is statically sensitizable iff there exists an input vector such that all the side inputs of the path are set to non-controlling values.

In the context of STA, a statically sensitizable path π is such that there exists a delay assignment for which π is a critical path (just make sure that all non-controlling side inputs arrive early). In other words, a statically sensitizable path is a true path. Note that a path that is not statically sensitizable can still be a true path, depending on the delay assignment.

2.5 Co-sensitization

Let π be a path $(y_1, x_2, y_2, x_3, y_3, \dots, x_n, y_n)$ from PI to PO , where x_k is the on-pin of the gate y_k , and y_{k-1} drives gate y_k . Let $\text{Side}(y_k)$ be the set of side inputs of gate y_k . Path π is statically co-sensitizable [1] iff there exists an input vector such that:

$$\prod_{k=2}^{k=n} \left[\left(\frac{\partial y_k}{\partial y_{k-1}} = 0 \right) \Rightarrow \left(\sum_{x_j \in \text{Side}(y_k)} \frac{\partial y_k}{\partial x_j} = 0 \right) \right] \neq 0 \quad (2)$$

Equation (2) means that whenever a gate y_k is not sensitized by its driver y_{k-1} , it is not sensitized by its side inputs either.

For a netlist made of elementary gates (e.g., AND, OR, NOT, XOR), this is equivalent to the following:

Definition 2. A path is statically co-sensitizable iff there exists an input vector such that whenever an output pin is set to a controlled value then the corresponding on-input is assigned a controlling value.

In the context of STA, a path π is not statically co-sensitizable iff for every input patterns, some inner term of equation (2) is 0. This means that for every input pattern, there exists a gate on path π that is controlled by a side input. Therefore a path that is not statically co-sensitizable is a false path. Note that the converse is false: a false path can still be statically co-sensitizable, depending on the delay assignment.

2.6 True and False Path Criteria

A path that is statically sensitizable is also statically co-sensitizable, producing the decision matrix shown in Table 1. A path that is co-sensitizable but not sensitizable is undecided, because its true or false nature depends on propagation delays.

Table 1. Decision criteria

Sensitizable	Co-sensitizable	Path Type
yes	yes	true path
yes	no	cannot occur
no	yes	undecided
no	no	false path

2.7 Direct Approaches

Verifying a generalized false path $G(P_1, \dots, P_n)$ in a brute-force manner consists of enumerating all the paths of G , and checking that they are all false. This is clearly not practical, as G can have an exponential number of paths w.r.t. to the size of the netlist.

A second approach consists of reducing G to a full path, and using formula (1) and (2) on that full path. The reduction is done by identifying the logic function between two anchors y_{k-1} and x_k . That function becomes a new gate between gates y_{k-1} and x_k .

Let $G(\{p_1\}, \dots, \{p_n\})$ be a *simple* generalized path, i.e., such that the anchor sets are singletons. We can cluster the logic between pin p_{k-1} and p_k to form a macro-gate. Figure 1 shows the macro-gate made of the logic contained in the intersection of the TFO (transitive fanout) of p_{k-1} and of the TFI (transitive fanin) of p_k . The inputs of the macro-gate are all the nets driving the side pins of the gates in the cluster (g_1, g_2, g_3 in the example). We can then apply the formula (1) and (2) using the macro-gates [8] [9].

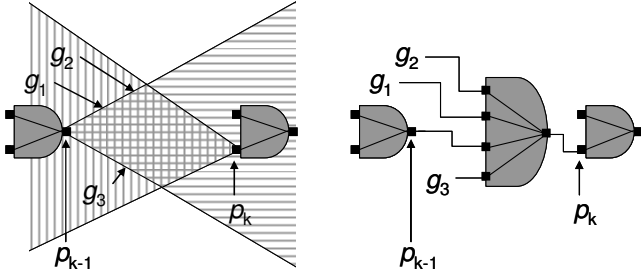


Figure 1. Macro-gate between pins p_{k-1} and p_k

Breaking down $G(P_1, \dots, P_n)$ into simple generalized paths is limited by the number of these paths, up to $|P_1| \cdot |P_2| \cdot \dots \cdot |P_n|$, which is still non-polynomial w.r.t. the size of G .

Checking (1) and (2) is anyway computationally expensive, in part because of the Boolean differences. The expression representing a Boolean difference of a gate g can be as large as $2 \cdot |TFI(g)|$, which produces a non-polynomial size expression for (1) and (2).

3. A Scalable Algorithm

This section presents an algorithm that overcomes the limitations due to the exponential number of paths in a generalized false path, as well as the non-polynomial size of expression (1) and (2). This

algorithm checks sensitization and co-sensitization of an exception in one pass, regardless of the number of paths, by checking satisfiability on a polynomial-size Boolean expression.

3.1 Preliminaries

Let us define the Boolean functions Sen and CoSen as follows. Given an input pattern I and a pin p , $\text{Sen}(p) = 1$ (respectively $\text{CoSen}(p) = 1$) iff there exists a path from PI to p that is statically sensitizable by I (respectively statically co-sensitizable).

Theorem 1. CoSen and Sen satisfy the following identities –in these identities, f denotes either Sen or CoSen:

$$f(p) = 1 \quad \text{for } p \in PI \quad (3)$$

$$f(\text{NOT}(x)) = f(x) \quad (4)$$

$$f(\text{XOR}(x_1, x_2, \dots)) = \sum_k f(x_k) \quad (5)$$

$$\text{Sen}(\text{AND}(x_1, x_2, \dots)) = \sum_k (\text{Sen}(x_k) \cdot \prod_{j \neq k} x_j) \quad (6)$$

$$\text{CoSen}(\text{AND}(x_1, x_2, \dots)) = \left(\prod_k x_k \right) \cdot \sum_k \text{CoSen}(x_k) + \sum_k (\neg x_k \cdot \text{CoSen}(x_k)) \quad (7)$$

Proof. Identity (3) is trivial. Identities (4) and (5) express that the output of a NOT or XOR gate is statically (co-)sensitizable iff any of its input pin is. This is because any change in the input values will be reflected at the output (i.e., the Boolean difference of a NOT or XOR output w.r.t. any of its input pin is 1).

Propagation of Sen and CoSen through a AND gate are different. Identity (6) states that the output pin is statically sensitizable iff there exist an input pin that is statically sensitizable and all the other pins are non-controlling (i.e., are 1). Identity (7) decomposes co-sensitization in two parts. If the AND gate is not controlled (i.e., all its inputs are 1), the output is co-sensitizable iff any of the input is. When the AND gate is controlled, only a controlling input pin (i.e., valued to 0) can carry the CoSen property to the output pin. \square

Sen and CoSen can be computed on the whole netlist for any input pattern I using identities (3-7). These identities capture everything needed for checking a timing exception but the notion of “being on a path”. Next Section shows how to efficiently build the Sen and CoSen formulas w.r.t a generalized path.

3.2 The Algorithm

Let $G(P_1, \dots, P_n)$ be an exception, where $P_1 \subseteq PI$ and $P_n \subseteq PO$. To verify G , we need to check that:

- either (i) no path of G is statically co-sensitizable, which implies that all paths in G are false (correct exception);
- or (ii) some path of G is statically sensitizable, which implies that some path of G is true (incorrect exception).

Let us restrict the definition of Sen and CoSen to G by requiring that the path leading to pin p is a subpath of G . Criteria (i) is true (correct exception) iff function CoSen is 0 on every pin of P_n . Criteria (ii) is true (incorrect exception) iff function Sen is 1 on at least one pin of P_n .

```

1. typedef {0, 1, Pin} Expr; // Expression

2. Expr BuildFun(f, G(P1,...,Pn)) {
3.   f.clear(); // Reset the function
4.   foreach side input p of G { f(p) = 0; } // Set side inputs
5.   foreach p ∈ P1 { f(p) = 1; } // Set f's value for start points
6.   for (k = 2; k <= n; ++k) {
7.     set<Pin> slice = Pk-1;
8.     foreach p ∈ Pk { BuildRec(f, p, slice); }
9.     foreach p ∈ slice { if (p ∉ Pk) f(p) = 0; }
10.  }
11.  return ∑p ∈ Pn f(p);
12. }

13. // Build f on the TFI of p up to the boundary Pk-1.
14. Expr BuildRec(f, p, slice) {
15.  if (!f(p).exist()) { // Not computed yet, because this
16.    slice.insert(p); // is a new pin discovered in Slice k
17.    if (p.isInput()) {
18.      f(p) = BuildRec(f, p.driver(), slice);
19.    } else {
20.      foreach x in p.gate().inputs() { BuildRec(f, x, slice); }
21.      f(p) = BuildForGate(f, p);
22.    }
23.  }
24.  return f(p);
25. }

26. // Build f on output pin y according to identities (4-7).
27. // Inputs are assumed to have a known formula expression
28. Expr BuildForGate(f, y) {
29.  Gate g = y.gate();
30.  vector<Pin> in = g.inputs();
31.  switch (g.type()) {
32.  case NOT:
33.  case XOR: return ∑x ∈ in f(x);
34.  case AND:
35.    if (f == Sen) // Static sensitization
36.      return ∑x ∈ in (f(x) · ∏q ∈ in, q ≠ x q)
37.    else // Static co-sensitization
38.      return y · (∑x ∈ in f(x)) + ∑x ∈ in (¬x · f(x))

```

Figure 2. Building CoSen and Sen w.r.t $G(P_1, \dots, P_n)$ in one pass

Figure 2 shows the algorithm **BuildFun** used to produce the Boolean formulas of CoSen and Sen w.r.t G . It takes an exception $G(P_1, \dots, P_n)$ as input, the function f to build (CoSen or Sen), and returns a Boolean expression. We implement f as a hash table that maps pins p onto the expression $f(p)$. Function f is cleared (line 3) before we start populating the hash table.

Function **BuildFun** returns (line 11) the OR of f 's values on the pins of P_n . Therefore criteria (i) is true (correct exception) iff running **BuildFun** with $f = \text{CoSen}$ returns an expression that is not

satisfiable (UNSAT, or antilogy). And criteria (ii) is true (incorrect exception) iff running **BuildFun** with $f = \text{Sen}$ returns an expression that is satisfiable (SAT).

Note that the expression, of type Expr, is built from pins of the original netlist plus some extra logic. Whenever a pin is used as an expression, it is understood that the expression is the Boolean function implemented by the netlist at that very pin. The expression returned by **BuildFun** is a single-output gate that can then be checked for satisfiability.

The algorithm is made of two main parts. One part (**BuildFun** and **BuildRec**) traverses the logic spanned by the exception $G(P_1, \dots, P_n)$ and incrementally build the function f . This traversal and assembly part is shared for the computation of both CoSen and Sen. The second part (**BuildForGate**) builds the function f for elementary gates according to identities (4-7) for CoSen and Sen.

3.3 Correctness

The generalized path $G(P_1, \dots, P_n)$ can be seen as a partitioning of the subgraph induced by G in $(n-1)$ slices. Slice k is made of the pins that lie on any path between P_{k-1} and P_k .

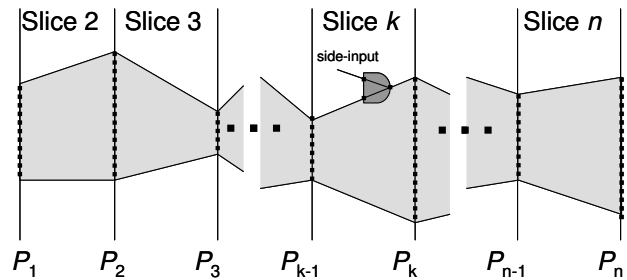


Figure 3. Slices of a generalized path $G(P_1, \dots, P_n)$

Function **BuildFun** computes the function f one slice at a time, from slice 2 to slice n (line 6-10). It calls **BuildRec** to traverse slice k in a DFS manner (line 8). Note that zeroing f on the side inputs (line 4) makes sure that **BuildRec** does not wander outside of the subgraph via a side input.

Once slice k has been processed, f 's value on slice k is reset to 0 except at the slice's outbound boundary P_k (line 9). The boundary values on P_k will then be used to build f for the next slice $k+1$. Zeroing slice k but its outbound boundary P_k forces the function f to 0 if **BuildRec** leaves slice $k+1$ without reaching P_k , i.e., if the path is not in G . Forcing f to 1 on the pins of P_1 (line 5) makes sure that f is built on the first slice (i.e., slice 2) with the correct value on its inbound boundary (i.e., P_1). This iterative slice-based construction of f guarantees that f will be 0 for any path that is not a subpath of G .

Without loss of generality, we assume the netlist to be made of AND, XOR, and NOT gates only. Function **BuildForGate** builds the function f at the gate level. When **BuildForGate** is called on the output pin of a gate, it is guaranteed that the function f has been built for its input pins (because of line 20). Gate NOT and XOR are processed (line 33) according to identities (4) and (5). An AND gate is processed for Sen (line 36) following identity (6). For CoSen, an AND gate is processed (line 38) following identity

(7): in the first component of the sum, y is assumed to be 1, which is equivalent to have all its input to 1, as formulated in identity (7); the second component of the sum is exactly as in identity (7).

3.4 Complexity

Having the formula f expressed as a hash table mapping pins onto a Boolean expression has the benefit of avoiding an initialization phase that would require setting all pins of the netlist to some undefined value.

The initialization phase, which forces all side-inputs to 0 (line 4), is non-trivial. One way is to reuse the same slice exploration idea, where the TFI of P_k is explored until reaching PI , or when a function value 0 is encountered. The overall complexity is however in $O(|N|)$. A better bound can be achieved if P_k 's TFI and P_{k-1} 's TFO are explored concurrently.

Since we visit each slice only once, the overall complexity of the traversal is linear with the size of the subgraph G . Let us measure the size of the Boolean expression for Sen and $CoSen$. When processing an n -input NOT or XOR gate (line 33), **BuildForGate** adds one n -input OR gate. When processing an n -input AND gate for function Sen (line 36), **BuildForGate** adds n n -input AND gates and one n -input OR gate –thus $O(n)$ gates and $O(n^2)$ pins. When processing an n -input AND gate for function $CoSen$ (line 38), **BuildForGate** adds one n -input OR gate, one $(n+1)$ -input OR gate, and $(n+1)$ 2-input AND gates –thus $O(n)$ gates and $O(n)$ pins. Since we can pre-process the netlist to expand any n -input AND gate into $\lceil \ln_2(n) \rceil$ 2-input AND gates, the size of the $CoSen$ and Sen formulas are in $O(|G| \ln |G|)$.

Of course the overall complexity of the verification algorithm is dominated by the UNSAT/SAT query that is called on the final expression, i.e., it is co-NP-complete for $CoSen$, and NP-complete for Sen .

3.5 Extensions

This section sketches out extensions to Section 3.2's algorithm.

3.5.1 4-valued Logic

One natural extension is to support the 4-valued logic semantics of Verilog [11], using the logic values 0, 1, Z, and X. The concepts of controlled and controlling logic extend to 4-valued logic with the same definitions. One way of extending the algorithm to 4-valued logic is to encode 4-valued logic signals with two Boolean signals.

3.5.2 Multi-driver Nets and Undriven Pins

Algorithm of Section 3.2 assumes that there is only one driver per input pin (line 18). With the support of 4-valued logic it is straightforward to support multi-driver nets using the pair of Boolean variables technique. Also undriven pins are simply considered as driven by the logic value Z.

3.5.3 Tristate

Once support for 4-value logic is established, we can add support for non-Boolean operators like tristates. A tristate is a cell $tri(in, en, out)$ that operates as shown in Table 2, following the Verilog semantics of `bufif1` [11]. Tristates are usually used to drive multi-driver nets.

Table 2. Tristate semantics

in	en	out
0	1	0
1	1	1
Z/X	1	X
0/1/Z/X	0	Z
0/1/Z/X	Z/X	X

The enable signal has the only controlling values, namely 0, Z, and X, with controlled value Z, X, and X respectively. This means that en is non-controlling iff it is 1. Thus the static sensitization property on a tristate is:

$$Sen(out) = Sen(en) + en \cdot Sen(in)$$

Similarly the static co-sensitization property on a tristate is:

$$CoSen(out) = CoSen(en) + en \cdot CoSen(in)$$

3.5.4 Debugging

When we prove that a generalized false path G is statically sensitizable, it implies that the exception is an incorrect constraint. In that case we would like to show the user an input pattern with a path that it sensitizes. Since a path is statically sensitizable iff Sen is satisfiable, we can build a sensitizable path from an assignment that satisfies Sen . If we preserve Sen 's expressions for all pins in another hash table as they are processed, we can evaluate the value of function Sen on any pin of G . Thus we can produce a path along which Sen is valued to 1 by starting from some pin of P_n that satisfies Sen and performing a DFS through the pins that satisfy Sen . The whole procedure, including evaluating Sen for the given input pattern whenever necessary, can be done in $O(|G|)$.

3.5.5 Clock-to-clock Constraints

SDC can specify false paths by referring to their clock domains. E.g., a constraint “set_false_path –from clk_1 –to clk_2 ” states that all paths from flip-flops ff_1 's clocked by clk_1 to flip-flops ff_2 's clocked by clk_2 are false. To verify this kind of constraints, we need to verify that whenever clk_1 and clk_2 are simultaneously active, paths from ff_1 's to ff_2 's are indeed false. This can be achieved using the same algorithm described in Section 3.2. Part of the complexity is in tracing the clocks so that the proper sets of flip-flops are extracted.

4. EXPERIMENTAL RESULTS

We used MINISAT [14] as the SAT solver. We took a set of industrial circuits and did two experiments (all runs are on a Linux Opteron 64 bits 2.1GHz with 8Gb RAM).

4.1 False Path Verification

The first experiment verifies the original SDC constraints coming with the design. We observed that breaking down SDC false paths into simple generalized paths (Section 2.7) increases the number of constraints by 100x-1000x, which makes that approach impractical.

Table 3. SDC verification

Design	#gate (1000)	#ff (1000)	#SDC	#Sen	#non-CoSen	%unvalid	CPU (mn)	#path/hour
ds_core	1003	54	1421	121	1296	8.5%	106.5	801
Kailash	3223	249	261	38	205	14.6%	88.6	177
modec_top	2352	132	1007	2	1005	0.2%	18.7	3231
sltd	415	21	107	1	106	0.9%	21.0	306
rosedale	1188	65	87	22	63	25.3%	36.5	143
PCHIP	2638	117	30	5	25	16.7%	12.2	148
SCUMA	474	26	240	17	123	7.1%	18.4	783
FTC	548	30	377	79	132	21.0%	10.1	2240
ARM	199	13	215	0	215	0.0%	13.6	949
avg.	1338	79	416	32	352	10.5%	36.2	975

The results are shown in Table 3. The throughput (#path/hour) varies quite a lot, since the computational cost depends a lot on the circuit structure and the SDC constraints. Overall it shows that we verify about 1000 SDC constraint per hour. This compares favorably with “100 paths per hour” [5], or with “up to 1000 paths on multi-million gate designs in a few hours” [6]. Note that the fraction of incorrect constraints can be as high as 25%.

4.2 Filtering False Paths Out

The second experiment consists of determining how many of the top most critical paths reported by STA turn out to be actually false. Filtering false paths out of a timing report produces a more accurate (less pessimistic) timing picture. This also generates new false paths that can be used for a better optimization.

Design	#gate (1000)	#ff (1000)	#path	Avg. path lgth	Sensitization				Co-sensitization				FP/hour	%FP
					#non-Sen	Mem (GB)	CPU (mn)	#path/hour	#non-CoSen	Mem (GB)	CPU (mn)	#path/hour		
ds_core	1003	54	5000	21	698	2.0	12.8	23478	62	2.4	12.8	23478	291	1.2%
Kailash	3223	249	4000	6	162	5.2	46.7	5143	10	5.3	46.7	5143	13	0.3%
modec_top	2352	132	2000	35	1113	4.0	38.9	3086	415	3.9	39.4	3042	631	20.8%
sltd	415	21	1200	18	245	3.7	10.0	7200	24	5.4	10.0	7200	144	2.0%
rosedale	1188	65	500	30	173	1.8	31.0	967	57	2.1	28.8	1043	119	11.4%
PCHIP	2638	117	500	16	0	6.4	12.2	2455	0	6.4	10.6	2842	0	0.0%
SCUMA	474	26	500	37	271	2.2	7.8	3857	108	3.4	7.8	3857	833	21.6%
FTC	548	30	100	2	100	0.4	5.6	1080	0	3.1	5.6	1080	0	0.0%
ARM	199	13	50	26	0	3.4	1.0	3000	0	3.4	1.0	3000	0	0.0%
avg.	1338	79	1539	21	307	3.2	18.4	5585	75	3.9	18.1	5632	226	6.4%

Table 4. Filtering false paths from timing reports

Table 4 shows the results. In that table, the average path length refers to the average length of the top #path most critical paths reported by STA. The table gives the CPU time in minutes to check all the reported paths for Sen and CoSen criteria –note that only criteria CoSen is necessary to filter false paths.

On average we check about 5600 paths/hour on a 1.3M gate design. About 14% of these paths are inconclusive, i.e., they are not statically sensitizable but they are co-sensitizable. Out of the critical paths reported by STA, we find about 226 false paths per hour. On average 6.4% of the paths reported by STA are actually false paths, with the fraction of false paths ranging from 0% to 21.6%. This is significant enough to justify new constraints to be added so that optimization can work on the actual critical paths.

5. CONCLUSION

This paper presented an efficient algorithm to check the correctness of a generalized false path G . It uses a recursive formulation of the static co-sensitization and sensitization criteria, from which two $O(|G| \ln |G|)$ -size Boolean formulas can be built. Checking for the satisfiability of these formulas determines whether the generalized false path is sensitizable or not co-sensitizable. This algorithm avoids path enumeration, and reduces

the size of the UNSAT/SAT problems the co-sensitization and sensitization queries boil down to.

This algorithm is about 10x faster than known industrial false path verifier, which makes it applicable to real-life multi-million gate designs. It also can be used to filter out false path from a timing report for a more accurate worst slack assessment, or to generate new false paths that can be used to further optimize the design.

Further work will focus on grouping SDC constraints so that some computational cost of the SAT solver can be shared. Also compressing an original set of SDC constraints is a natural extension of this work.

6. ACKNOWLEDGMENTS

The author would like to thank Sneha Saurabh, without whom this paper would not have been possible. Sneha’s early insights into false path verification, and how to extend verification to generalized false paths, ignited relentless and creative discussions that eventually led to the method described here.

7. REFERENCES

- [1] S. Devadas, K. Keutzer, S. Malik, “Computation of Floating Mode delay in a combinational circuits: Theory and Algorithms”, IEEE Trans. on CAD Design, Dec. 1993
- [2] H.-C. Chen, D. Hung-Chang Du, “Path Sensitization in Critical Path Problem”, IEEE Trans. on CAD of Integrated Circuits and Systems, Feb. 1993.
- [3] F.S. Marques, R.P. Ribas, S. Sapatnekar, A.I. Reis, “A New Approach to the Use of Satisfiability in False Path Detection”, Proc. of GLSVLSI’05, April 2005.
- [4] *Using the Synopsys Design Constraints Format*, Version 1.7, March 2007.
- [5] Averant, White Paper, DesignCon, 2007.
- [6] Fishtail, personal communication, 2007.
- [7] Silva, et al. “Realistic Delay Modeling in Satisfiability-Based Timing Analysis”, Proc. of ISCAS ’98, pp. 215-218, May 1998.
- [8] H.-H. Chao, R. Razdan, A. Saldanha, “Method and Apparatus for Critical and False Path Verification”, US Patent 6714902, March 2004.
- [9] M. Ringe, T. Lindenkreuz, E. Barke, “Path Verification Using Boolean Satisfiability”, Proc. of DATE’98, pp. 965-966, 1998.
- [10] A. Ghosh, K. Keutzer, S. Devadas, *Logic Synthesis*, McGraw-Hill, 1994.
- [11] IEEE 1364-2001, *IEEE Standard Verilog Hardware Description Language*, Section 7-8, 2002.
- [12] K. Yang, K.-T. Cheng, “Efficient Identification of Multi-Cycle False Path”, ASPDAC, pp. 360-365, 2006.
- [13] R. Drechsler, S. Eggersglüß, G. Fey, A. Glowatz, F. Hapke, J. Schöffel, D. Tille, “On acceleration of SAT-based ATPG for industrial designs”, IEEE Trans. on CAD of Circuits and Systems, 27, pp. 1329–1333, 2008.
- [14] N. Eén, N. Sörensson, “An Extensible SAT-Solver”, *SAT*, pp. 502–518, 2003