Identifying Untestable Transition Faults in Latch Based Designs with Multiple Clocks

Manan Syal†, Sreejit Chakravarty†† and Michael S. Hsiao†
† Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA.
†† Intel Architecture Group, Intel Corporation, Santa Clara, CA.

Abstract
This paper presents a novel technique to identify functionally untestable transition faults in latch-based designs with multiple clock domains, bringing to light unaddressed issues related to untestable fault identification in such design environments. We also introduce and provide a solution to a new variant of untestability analysis wherein "architectural constraints" are absorbed during the analysis. We give our tool the capability of handling transition faults resulting from defects of varying sizes, and evaluate our tool for various industrial circuits. The proposed algorithm is compared with a state-of-the-art sequential ATPG tool, and our method has shown much better performance both in the context of scan ATPG and functional test development. Results indicate that the proposed technique identifies considerably more untestable transition faults than those that can be deduced from the knowledge of untestable stuck-at faults. Additional insights from our results point to a greater need to eliminate untestable transition faults as compared to stuck-at faults, for more efficient test pattern generation and accurate coverage computation.

1. Introduction

Transition faults are defined to be caused by defects that cause a logic value transition (0→1 transition or 1→0 transition) on a net to be delayed such that the transition does not reach a primary output (PO) or scan flip-flop. If the defect causes the 0→1 (1→0) transition to be delayed, then the fault is called slow-to-rise (slow-to-fall) or STR (STF) transition fault. Generally speaking, a transition fault requires a pair of test patterns (T1, T2) to be tested. T1 is required to set the fault-site to the initial value, and T2 is needed to launch the appropriate transition on the fault-site and also to detect the fault-effect at a PO. For full-scan designs, T1 is a single vector, while for partial scan and non-scan designs, T1 may be a sequence of test vectors. Scan-based transition tests can be applied in three different schemes: Broadside [1], Skewed-Load [2], and Enhanced-Scan [3]. In the context of stuck-at faults, to detect a STR (STF) transition fault, test sequence T1 must excite a stuck-at-1 (stuck-at-0) on the fault-site, and T2 must detect a stuck-at-0 (stuck-at-1) on the fault-site.

Considerable work has been reported on efficient test-pattern generation [4] [5] [6] for transition faults, test-data volume reduction for transition tests [7] [8] etc, but no significant contributions other than the recent work by Chen et al. [9] have been made to identify functionally untestable transition faults. Moreover, all previous work on un-testability analysis, for stuck-at [10][11][12][13] as well as transition faults [9], assumes the design to be single-clock and flip-flop based. In this paper, we address the problem of identifying un-testable transition faults in multi-clock latch-based designs. Many commercial designs have these design traits and increased performance is contributing to an increase of such design characteristics. Untestability analysis of faults in such designs has not been addressed previously. We study and highlight some challenges and issues that complicate untestable fault analysis in such design environments.

Furthermore, traditionally, analysis of transition faults assumes that the defect size is such that the faulty behavior of the transition fault persists for only one clock cycle. In order to target a wide range of delay defects, in the kind of designs we are targeting, we are interested in transition faults with delays that could persist for one more than one clock cycle. In order to parameterize the defect size for such variable delay transition faults, in addition to the location and kind of transition fault we also assign an integer value denoting the duration for which the fault effect can persist. The need for specifically addressing the untestability problem for variable delay transition faults is highlighted by the following theoretical result which we establish in Section 3. We show that a transition fault, at node X, with one cycle delay could be untestable but if the delay effect persists for more than one cycle then the transition fault at X may become testable.

In the context of scan ATPG, the ATPG engine is often constrained to avoid design contention. Other constraints could result from initialized or unreachable states, DFT modes etc. In the context of functional testing, if we limit the untestability analysis to a fixed number of time frames we often cannot identify faults that require an unreachable state be visited in order for the fault to be detected. It is important to identify faults that can only be detected if such constraints are violated and classify them as untestable faults. The idea of absorbing constraints, whatever be the source, has not been studied in the context of identifying untestable faults. In Section 4 we show how such constraints can be absorbed during the untestability analysis process.

* This work was performed while the first author was at Intel, Santa Clara from May '03 – August '03. The work is also sponsored in part by a grant from Intel and by NSF grants CCR-0196470 and CCR-0305881.
The composite untestability analysis algorithm is described in Section 9. We compare the usefulness of this algorithm with an ATPG tool in two different contexts: (i) scan ATPG and coverage reporting; and (ii) functional fault exclusion. We show that, for a number of industrial designs, in both of these contexts the approach suggested outperforms an industrial sequential ATPG tool and helps in reducing the total time to identify such untestable faults. We also show that a large number of transition faults are indeed untestable thereby highlighting the need to identify such faults for improved coverage analysis.

Many researchers believe that identification of untestable transition faults can be easily performed using the knowledge of stuck-at untestable faults alone. We show that this is certainly not the case for the industrial test cases we considered. If we rely only on the knowledge of untestable stuck-at faults a very large number of untestable transition faults cannot be identified thereby showing the need to develop techniques specifically for identifying untestable transition faults.

The rest of the paper is organized as follows. Section 2 explains logic implications and symbolic simulation as they form the core of our untestable fault identification engine (implications and symbolic simulation are used both for architectural constraints and for the proposed untestable transition fault analysis). Section 3 describes the computation model that we use to identify untestable faults in multi-clock, latch based designs. Here we also explain the significance of defect size w.r.t. identification of untestable transition faults. In section 4 we explain the use of architectural constraints in identification of untestable faults. Section 5 describes the proposed technique for per-fault untestable transition fault identification and we explain how this technique is used for latch-based designs with multiple clocks. Section 6 discusses the challenges and complexities associated with identification of untestable transition faults in latch based designs with multiple clocks. In section 7 we describe the overall algorithm. Section 8 discusses experimental results, and section 9 concludes the paper.

2. Preliminaries

2.1 Logic Implications

Logic implications illustrate the effect of asserting logic values in a circuit. Broadly, implications can be classified into direct, indirect and extended-backward implications [11]. While, direct implications can be easily enumerated by traversing the immediate fan-in and fan-out of a gate, indirect and extended backward (E-B for short) implications require the use of circuit simulation. In order to understand the concept of implications, consider the circuit shown in Figure 1. Let us consider the implications of gate \( e = 0 \). Now, it can be easily seen that the direct implications of \( e = 0 \) are the set \( \{ d = 0, c = 0, f = 0, g = 0 \} \). It can also be seen from Figure 1 that even though \( f = 0 \) and \( g = 0 \) do not individually imply any value on gate \( i \), however, together, they imply \( i = 1 \). This is an indirect implication, and is learnt through circuit simulation. Also, since \( d = 0 \) is an unjustified gate in the implication list for \( e = 0 \), it forms a candidate for the application of E-B implications.

To perform E-B implications on \( d \), first, gate \( a \) is set to logic value 0, and the circuit is simulated with the implications of \( e = 0 \). This yields two new assignments, \( j = 1 \) and \( l = 1 \). Then, the other input of \( d \), i.e. gate \( b \) is set to logic 0, and the circuit is simulated as before.

Simulation again yields new assignments i.e. \( h = 1 \), \( k = 1 \) and \( l = 1 \). Since the assignment \( l = 1 \) is common between the learnt values during the simulation of \( a = 0 \) and \( b = 0 \), it is added as an E-B implication to the implication list for \( e = 0 \).

It is evident that enumeration of indirect implications of any \( \{ \text{gate} = \text{value} \} \) requires one pass of logic simulation for the circuit. However, identification of E-B implications may require multiple passes of circuit simulation. For large industrial circuits, this process of E-B implication enumeration can thus prove to be a performance bottleneck. Similar observation has been reported in [14] and in the results of [13]. Thus, in our untestable fault identification engine, in order to keep execution time within reasonable bounds, we choose not to perform extended-backward or E-B implications.

2.2 Symbolic Simulation

We use symbolic simulation as a means to identify uncontrollable nets (i.e. nets that cannot be excited to a particular logic value) in a design before untestable fault identification is invoked. The need for this symbolic simulation engine is motivated from the fact that uncontrollable nets imply a lot of untestable faults (if a net \( n \) is uncontrollable to logic value \( \nu \), then not only are both STR and STF transition faults on that net untestable, but other transition faults that require \( n = \nu \) as a necessary condition for their detection would also be untestable). The framework and the symbolic values that we use to perform symbolic simulation is the same as that used in [14]. We use 11 symbolic values as in [14] because these symbols provide support for characterizing nets that can take high impedance values (i.e. Z).

3. Computation Model

In this section, we describe the computation model that we use in our untestable fault identification engine for latch based designs with multiple clocks. First, consider the model of a sequential circuit as shown in Figure 2(a).
The sequential circuit is modeled as a combination of sequential elements and a combinational logic $C$. The combinational logic $C$ is fed by primary inputs (PI) and the present state variables (P(t)). $C$ generates logic values on primary outputs (PO) and also the next-state variables (N(t)) which are fed into the combinational logic in the next cycle as present state variables through the sequential elements. Figure 2(b) shows the same sequential circuit, unrolled in k-time frames (0 – (k-1)). In this unrolled version, also called the iterative logic array (ILA), k-copies of the combinational logic are connected to each other in such a way that the next state variables of any copy $j$ ($N_j$) feed the present state variables of copy $j+1$ ($P_{j+1}$). This ILA unrolling in k-time frames with k-combinational copies assumes that the sequential elements are flip-flops that change state at a particular clock edge. Also, the ILA shown in Figure 2(b) corresponds to a circuit that has a single clock.

In latch based designs, some latches are sensitive to the positive level of the clock, and some latches are sensitive to the negative level (this is positive level of the clock, and some latches correspond to a circuit that has a single clock. clock edge. Also, #the elements combinational copies assumes functionality as flip-flop representing logic C in phase and C, representing logic (say $f,P,+,+.$).

Consider a design with $2k$ combinational copies of the circuit, $k$-corresponding to each phase of the clock). Thus, the ILA must contain two copies of C (one corresponding to each phase of the clock). The k-frame ILA for a single-clock, latch based design now would consist of $2^k$ combinational copies of the circuit, and appear as shown in Figure 3. Here $C_1$ and $C_2$ represent two copies of the combinational logic $C$, with $C_1$ representing logic $C$ in phase $\Phi_1$ and $C_2$ representing logic $C$ in phase $\Phi_2$.

Now let us consider designs with multiple clocks. Consider a design with three clocks $clk_1$, $clk_2$ and $clk_3$, the waveforms for which is shown in Figure 4. The three clocks divide the functionality of the design into four unique phases, $\Phi_1-\Phi_4$, corresponding to the four unique combinations of $clk_1$, $clk_2$, and $clk_3$. For this design, each time frame in the ILA would have 4 copies of the combinational logic of the circuit, one corresponding to each phase. Again, the functionality of each copy can potentially be different from any other copy in the same time frame.

So, generally, if the clocks divide the design into $n$-phases, then the ILA for the sequential circuit would have $n$-copies of the combinational logic for each time frame. Therefore, a k-frame expansion of the sequential circuit would have a total of $kn$ copies of the combinational logic. Figure 5 shows the generic ILA for an n-phase design. This is the circuit model that we use in our engine for untestable-fault identification. We first identify the number of unique phases for the design (based on the clocking waveforms), and create an ILA for the circuit based on this generic model.

Before describing the proposed technique to identify untestable transition faults, we need to understand the significance of the defect size (resulting in transition faults) with respect to untestable fault identification. Defect size determines the number of clock cycles for which a transition fault would exist once the fault has been excited. If the fault effect is not detected or latched in a memory element within this period, the faulty-effect of the slow transition would be lost (the faulty-net would achieve its final value in the transition). We call this period sustain count of the fault. To illustrate transition faults resulting from differing defect sizes, two transition faults are shown in Figure 6: the size of the defect resulting in the first transition fault $t_1$ is two clock cycles (i.e. once excited, the delayed transition attains its final value within two clock cycles), while the second defect results in the other transition fault, $t_2$, the size of which is one cycle.
It is important to understand that untestability analysis on transition faults is directly dependent upon the defect size. Defect size determines the least number of cycles over which the transition fault must be analyzed for untestability, once the fault is excited. For example, fault \( t_1 \) in Figure 6 must be analyzed for at least two clock cycles. If it is not possible to latch the fault effect for \( t_1 \) cycles, it must be analyzed for at least two clock cycles once it is excited. Since the fault was excited in time frame \( t + 1 \), it must be analyzed in time frames \( t + 1 \) and \( t + 2 \). In time frame \( t + 2 \), by setting \( x = 1 \), the faulty effect of the STR transition fault can be sustained. Also, in frame \( t + 2 \), \( Q = 1 \) (\( c = 1 \) in \( t + 1 \)), which would propagate the fault effect to the output \( y \), resulting in fault detection.

In our analysis, since we deal with latch based designs with multiple clocks, we measure the size of each defect (i.e. sustain count) in terms of the maximum number of phases (as opposed to number of cycles for flip-flop based designs) for which the faulty effect of the transition fault would exist. We ensure that untestability analysis on transition faults is performed while accounting for this size parameter. This size parameter, sustain count, is programmable, and can take any value, enabling us to handle transition faults of different sizes. In our analysis, we set this parameter equal to the number of unique phases in each clock cycle. However, our model and analysis can handle defects of any size.

4. Architectural Constraints

Architectural constraints are functional constraints that specify a group of nodes in the circuit to have certain logic values in order to prevent the occurrence of conditions that violate the design's specifications or violate some design rule. For example, a functional constraint may be imposed on some nodes in the circuit to prevent the state machine from going into a particular state or for prevention of bus-contention, etc. The term node is used in the above description because these architectural constraints can be specified on circuit components at any level of the design hierarchy (at register-transfer-level or RT-level, nodes would be components like adders or multipliers, while at gate-level nodes would correspond to logic gates). We extract such architectural constraints from the RT-Level description of the design and map these constraints to gate-level constraints. Both of these functions (extraction and mapping of constraints) are performed by internal tools. Once a constraint is available at the gate-level, we convert it to an AND-OR graph, and perform the following fault-independent analysis to find untestable faults:

/* untestable-fault identification with arch. constraints */

1. Associate all nodes in the graph with an unknown value (X). Associate every node in the graph with a set \( S_{\text{untest}} \). This set associated with every node \( n \) identifies the set of untestable faults identified at \( n \). Initially, for all nodes, \( S_{\text{untest}} = \phi \).
2. Since the constraint must be satisfied, associate a value of "true" (logic value 1) with the root R.
3. Propagate this value along the graph (keeping track of its polarity) in a depth-first fashion till you hit a leaf-node (LN). LN corresponds to a gate in the circuit.
4. While some leaf-node is assigned to value X:
   4.1. Assign the propagated value (v) to the leaf, and imply \( \{ LN = v \} \). Identify the set of faults \( Set \), in the circuit that are un-excitible or unobservable with

---

*Paper 36.2*
{LN_1 = v}. This procedure is similar to the process of finding untestable and unobservable nets used in FIRE \cite{10}. Return Set_1 to the parent P of LN_1.

4.2. If (Set_1 is the first set returned to P)

\[ S_{\text{untest}} = S_{\text{untest}} \cup \text{Set}_1 \]

Else

(If value at P = non-controlling for P)

\[ S_{\text{untest}} = S_{\text{untest}} \cup \text{Set}_1 \]

Else //value is controlling value for P

\[ S_{\text{untest}} = S_{\text{untest}} \cap \text{Set}_1 \]

4.3. Traverse the un-traveled path of P (path with children associated with value X), and propagate the value associated with P along this path in a depth-first fashion till you hit a leaf node (LN).

5. Go to step 4.

6. S_\text{untest} associated with R identifies transition faults untestable if R = true (i.e. if the constraint is satisfied). Since the constraint must always be satisfied, S_\text{untest} identifies truly untestable transition faults.

Consider an example to understand the algorithm described above.

Assume that an architectural constraint is:

\[ F = ((a \text{ OR } b) \text{ AND } c) \text{ OR } (-p \text{ AND } q) \]

Here a, b, c, p and q represent some gates in the circuit and \(-p\) refers to the logical complement of p. The AND-OR graph for F is shown in Figure 8.

![Figure 8: Graphical representation of architectural constraint](image)

The values associated with each node represent the value as propagated from the root (OR) node, and along with every node an operator (\(\cup\) or \(\cap\)) identifies the operation performed on the set(s) of faults propagated to a node from its successors. Now, according to the algorithm, the graph is first initialized, i.e. the value associated with each node in the graph is set to \(X\), and the untestable fault set S_\text{untest} at every node is set to null or \(\emptyset\).

Now, the root node (OR node at the top of the graph) is associated with a logic 1, and the graph is traversed from the root in a depth-first fashion. Assuming that the left branch is always chosen first in this depth-first traversal, the first leaf-node that we hit is \(a\). The value propagated to node \(a\) is logic 1. We assign node \(a\) to logic 1, and imply the assignment \{a = 1\} to identify Set_1 as the set of transition faults untestable if \(a = 1\). This set is returned to the parent of \(a\), which is an OR node. Since this is the first set returned to this OR node, S_\text{untest}\_OR \cup Set_1.

Next, we traverse the other X-path from this OR node, and reach node \(b\) with a logic value of 1. Again, we imply \{b = 1\} and find Set_1 as the set of faults untestable with \(b = 1\), and return this set to the parent (OR) node. This is not the first set returned to this OR node from any of its successors. Since the value associated with this OR gate (logic 1), is the controlling value for the node, we perform an intersection of Set_1 with S_\text{untest}\_OR, i.e. S_\text{untest}\_OR \cap = Set_1. This S_\text{untest}\_OR is returned as Set_1 to its parent, i.e. the AND node.

Since this is the first set of untestable faults returned to the AND node, a union operation is performed at the AND node, i.e. S_\text{untest}\_AND \cup = Set_1. The next leaf node reached according to the depth-first traversal would be node c, with a logic value of 1. So, the assignment \{c = 1\} is implied to identify the set of faults untestable with gate c set to 1. This set is returned to the parent of c, i.e. the AND node. This is not the first set returned to this node, so the next set operation depends upon the logic value associated with this AND node. Since logic 1 is non-controlling for AND gate, a union of S_\text{untest}\_AND is performed with the untestable fault set returned by node c.

This process of set intersection or union is continued for every node until all leaf nodes are assigned some logic value, and when we return to the root (OR) node an intersection of the set of untestable faults present at its children is performed (since the root in this case is an OR node) to obtain the final set of untestable transition faults.

5. Our Technique

In this section we describe the per-fault methodology we propose to identify untestable transition faults. Transition faults are of two types: slow-to-rise (STR) and slow-to-fall (STF). In order to test a transition of type \(v \rightarrow v'\) on some net \(n\) (here \(v \in \{0, 1\}\) and \(v'\) represents the logical complement of \(v\)), we must generate two test sequences. The first vector sequence must excite a stuck-at-\(v\) fault on \(n\) by the last vector of the sequence, and the second sequence must excite the stuck-at-\(v\) fault on \(n\) by the first vector of the second sequence and propagate the fault effect to an observable point. For example, for a STR (transition 0\(\rightarrow\)1) fault on some net \(n\), a test-pattern pair \{T_1, T_2\} must be applied such that the first test sequence \(T_1\) excites a stuck-at-1 on \(n\), and \(T_2\) detects a stuck-at-0 fault on \(n\). Since the problem of transition fault detection can be modeled as the problem of detecting stuck-at-faults, we can use theories behind techniques used for identification of untestable stuck-at-faults to identify untestable transition faults. In order to understand and appreciate our methodology, let us first look at the single-fault-theorem (SFT) proposed in \cite{13} and the theorem proposed in \cite{13}. According to the single fault theorem, if a single stuck-at fault injected in the last time frame of k-frame unrolled sequential circuit (ILA of size \(k\)) is found to be combinatorially untestable, then the fault would be sequentially untestable as well. The theorem proposed in \cite{13} relaxed the condition of fault-injection.
This theorem allowed fault-injection into any time frame of the ILA as opposed to the last time frame as in the single-fault-theorem. According to the statement of this theorem, a fault $f$ injected into any time-frame of a $k$-frame unrolled sequential circuit is found to be combinationally untestable if:

i) $f$ does not recombine with its copy in any higher time frame; or

ii) If recombination occurs, the combined fault-effect gets blocked before it can reach any observable point.

In the paper [13], the authors used the center of the ILA as the frame for fault-injection. In both SFT and in the theorem in [13], the state-elements in the lowest time frame of this ILA are assumed to be fully controllable (like primary inputs) and the state elements in the last time frame are considered as fully observable (and hence behave as primary outputs).

It is easy to see that SFT puts more emphasis on fault-excitation than the theorem in [13] (it justifies the condition for fault-excitation over $k$-time frames, while the theorem in [13] justifies the excitation condition over $k/2$ frames). However, the theorem in [13] attempts fault-propagation over $k/2$ frames, while SFT attempts fault propagation over just one (i.e. the last) time frame. Thus, some faults identified as untestable by SFT would be missed by the theorem in [13], while some faults identified as untestable by [13] might not be identified as untestable by SFT. This relationship between SFT and the theorem in [13] in terms of the faults identified as untestable by them is also shown in Figure 9.

![Figure 9: Unstable faults identified by SFT and [13]](image)

Our goal is to identify the union of the sets shown in Figure 9. So, we propose a technique that can be used to identify faults identified as unstable by SFT and also those identified as unstable by [13]. In the following analysis, we would first explain the methodology for flip-flop based designs with a single clock. Later, we would explain how the analysis is performed for latch based designs with multiple clocks. Also, in the following analysis, for simplicity, assume that the defect size results in a transition fault of one cycle.

Let us first consider the single-clock, flip-flop based design environment. In our technique we first unroll the sequential circuit into $k$-time frames ($0$-($k$-1)). Now, we divide the analysis into two parts: fault-excitation and fault-propagation. In the fault-excitation part, we inject the fault in the last time frame, and justify the condition for fault excitation over $k$-time frames. Since we deal with transition faults, fault excitation for a transition fault of type $v \rightarrow v'$ refers to setting the fault site to logic value $v$ in time frame $k$-2 and to logic value $v'$ in time frame $k$-1. These two conditions are then justified using implications (unlike single-fault-theorem, we do not use combinational ATPG due to the high cost associated with it for large circuits). So, by injecting the fault in the last time-frame we give as much emphasis to fault-excitation as in SFT.

In the fault-propagation part, we shift the window of analysis by $k$-units, and re-inject the fault into the lowest time frame, i.e. time frame 0 of the ILA. We then try to propagate the fault-effect (according to the theorem proposed in [13]) to any primary output or to the sequential elements in the last time frame. If the fault-effect gets blocked before it can be observed at any primary output or at the sequential element boundary, the fault is concluded as untestable. Thus, by propagating the fault-effect over the length of the entire ILA, we give fault propagation at least as much emphasis as given in [13].

So, by dividing the analysis into excitation and propagation phases, and by shifting the window of analysis over the ILA during the fault-propagation phase, we incorporate the advantages of both SFT and the theorem in [13] into our approach. These two parts of analysis are also shown in Figure 10 for a transition fault "n STR".

![Figure 10: Illustration of our technique for a STR](image)

Let us now understand how the technique explained above is applied to latch based designs with multiple clocks. For simplicity, assume that the clocking scheme for a design is such that there are two unique clock phases, $\Phi_1$ and $\Phi_2$. The ILA of size two (for simplicity) for such a design is shown in Figure 11.

![Figure 11: Two-frame ILA for a two-phase design](image)

Consider some transition fault $n$ STR. This fault would be untestable only if it is untestable in both the phases, i.e. $\Phi_1$ and $\Phi_2$. Therefore, we must first attempt to excite the fault in $C_1$ (which corresponds to $\Phi_1$) of the last time
frame, i.e. frame 1. This is done by setting \( n = 0 \) in \( C_2 \) of frame 0, and \( n = 1 \) in \( C_1 \) of frame 1. If the fault can be excited, then we record the excitation phase. Then we attempt to excite the fault in \( C_2 \) of frame 1. This is done by setting \( n = 0 \) in \( C_1 \) of frame 1, and \( n = 1 \) in \( C_0 \) of frame 1. Again, if the fault can be excited, we record the excitation phase. Then, we proceed to the propagation part of the analysis. Here, corresponding to each valid excitation phase, we re-inject the fault in lowest time frame, and attempt to propagate the fault effect to a primary output, or to the latch elements corresponding to \( N_l \). If the fault effect gets blocked before it can reach any of these observable points, we conclude the fault as untestable.

6. Challenges in Latch-Based Designs with Multiple Clocks

In this section we address some issues related to multiple clock designs that add to the complexity of untestable fault analysis. Firstly, it is easy to see that the ILA expansion for latch-based designs with multiple clocks is more intensive in memory as compared to flip-flop based single-clock designs. As opposed to one combinational copy per-time frame, the ILA expansion for multi-clocked designs has \( m \)-copies of the combinational circuit in each time frame, where \( m \) is the number of unique phases per clock cycle. This additionally adds to the computational complexity of any algorithm used for untestable fault identification because each value justification and fault propagation must be performed through more combinational logic as compared to flip-flop based designs. One alternative to having multiple copies of the entire combinational logic in each phase is to partition the circuit in regions according to the phases in which each region is functional. However, this partitioning may not be too beneficial because most of the regions in circuits are active or affected in more than one phase. So, each of these combinational regions would have to be replicated in each phase in which they are functional. In the worst case, this partitioning analysis might end up in replicating the entire combinational logic in each phase, which is no different from the computational model used. Thus, this pruning of logic based on phases can potentially add to the computational complexity of the problem without significantly reducing the memory requirements or time-intensiveness of untestability analysis.

Another characteristic that adds to the computational complexity for such designs results from the fact that we must perform untestability analysis (both excitation and propagation steps) for each unique phase in the design. If the transition fault is not testable in all the phases, then only can the fault be concluded as untestable. This is true for any fault-model (even stuck-at faults) in designs with multiple clocks. So, while for flip-flop based designs with single clock, a fault must be excited and propagated \textit{just once} before it can be declared as untestable, for latch based designs, a fault must be excited and propagated \textit{num_phases} number of times, where \textit{num_phases} is the number of unique phases for the design.

Also, the size parameter of a transition fault (sustain count) adds to the computational complexity of untestable fault analysis for designs with multiple clocks. For the same defect of size (say) \( n \)-cycles, in a flip-flop based design with single clock, the number of combinational copies across which the fault effect must be analyzed, by sustaining the fault effect, is \( 'n' \). However, for latch-based designs with multiple clocks, the fault effect must be sustained for \( n \times (# \text{ unique phases per clock cycle}) \). This clearly indicates the complexity of untestable fault identification in such circuits.

Despite the computational overhead, in the past, branch-and-bound algorithms (for example ATPG have been applied to untestable fault identification for single-clock flip-flop based designs. However, due to the reasons mentioned above, ATPG based algorithms would prove to be extremely expensive in both time and memory in the context of untestable fault identification for most of the industrial such circuits. Our tool built on implications, symbolic simulation and extraction of architectural constraints is better suited for untestable fault identification for such circuits.

7. Overall Algorithm

Before we describe the overall algorithm, we define a \textit{dominance point} in the following way:

\textbf{Dominance Point:} A dominance point is defined as the nearest fan-out stem, latch or primary output in the fan-out cone of the fault site.

For a fault to be testable, it must be propagated to its dominance point. In other words, if a fault cannot be propagated to its dominance point, then it cannot propagate to a primary output or to any other observable point in the net-list.

The complete algorithm for identification of untestable transition faults is discussed below:

/* Identification of Unstable Transition Faults */
1. Perform symbolic simulation to identify constants
2. Use untestable stuck-at faults and filter out some transition faults. (If \( n \) stuck-at-0(1) is unexcitable, then both transition faults on the net \( n \) (STR and STF) would be untestable; if \( n \) stuck-at-0(1) is not propagatable, then a single transition fault, i.e. STR(STF), on \( n \) is untestable)
3. Identify number of unique clock phases as \( p \)
4. Unroll the circuit into \( k \)-time frames (each frame has \( p \), copies of combinational logic)
   //For our experiments, \( k = 3 \)
5. Extract architectural constraints
6. Identify untestable faults using architectural constraints as explained in section 4, and remove these faults from the fault-list.
7. For each fault \( f \) in the filtered fault-list (per-fault analysis):
   A. Part-I (excitation):
      i) Identify clock-phases \( (p_1, p_2, \ldots) \) in which \( f \) can be excited
   B. Part-II (propagation):
      For each excitation phase \( p_i \):
      i) Inject the fault in the lowest time frame, and associate it with excitation phase \( p_i \)
      ii) Find the necessary condition(s) for the fault. Necessary conditions: conditions fault-propagation to dominance point.
      iii) Imply the necessary conditions
      iv) If no conflict exists in implications, attempt propagation of the fault effect to an observable point (i.e. a primary output or to any latch in the last time frame).
      v) If fault-effect reaches an observable point, declare fault as "not-untestable" and break.
      vi) Else, sustain fault-effect in next phase (account for defect size) if parameter sustain-count allows sustaining of fault.
      vii) Go to step ii)

8. Experimental Results

In this section we present results of experiments performed on some industrial circuits. The algorithm explained in Section 7 was implemented in C++, and experiments were run on a 2.0 GHz Pentium workstation, having 512 MB RAM, with Linux as the operating system. We present results that illustrate the effectiveness of our untestable fault identification tool in two testing scenarios: functional testing and scan based testing.

At high-level, functional test-pattern-generation or FTPG can be viewed as a two step process: In the first step, which is called fault exclusion, faults that violate architectural constraints and faults that are functionally untestable are excluded from the fault list; in the second step, functional patterns are generated for the remaining faults.

Two approaches can be used for fault-exclusion:

a) Through sequential ATPG (S-ATPG). If S-ATPG can generate tests for say \( x\% \) of the faults in the fault list, and if \( x\% \) fault-coverage is acceptable, then the remaining \((100-x)\%\) of the faults for which patterns could not be generated can be excluded from FTPG. Irrespective of whether these \((100-x)\%\) faults are just hard-to-detect or untestable, they can be excluded because the remaining \( x\% \) faults, that are testable, provide high fault coverage;

b) Through explicit identification of untestable faults. Here a tool such as the one described in this paper can be used to explicitly identify untestable faults, and exclude them from FTPG.

Table 1 reports results that indicate the performance of S-ATPG in the context of fault exclusion. Here, column 1 lists out a few industrial circuits that were used for our experimental study, and column 2 lists out the number of transition faults targeted in each circuit. In column 3 and 4 we report the number of faults detected using S-ATPG and the time taken to detect these faults. Finally column 5 lists out the \% of transition faults excluded by S-ATPG (note that these faults may be detectable if the number of test-cycles or the number of backtracks used in sequential ATPG is increased). We used an in-house transition fault sequential ATPG for this set of experiments, and analyzed each fault over 32 cycles with backtrack limit set to 2000. It can be seen from Table 1 that ATPG excludes more than 50% of the faults in most of the cases. This would mean that if sequential ATPG were used to exclude faults from FTPG, then we would be targeting a functional fault-coverage of barely around 50%, which is definitely not acceptable. Not only that, the time taken by S-ATPG to analyze the entire fault list is high. Thus, using sequential ATPG for fault-exclusion does not seem to be a practical choice.

<table>
<thead>
<tr>
<th>Circuit</th>
<th># detected</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ckt 1</td>
<td>1180</td>
<td>4.5 hrs</td>
</tr>
<tr>
<td>Ckt 2</td>
<td>9295</td>
<td>3 hrs</td>
</tr>
<tr>
<td>Ckt 3</td>
<td>17222</td>
<td>2.5 hrs</td>
</tr>
<tr>
<td>Ckt 4</td>
<td>1820</td>
<td>4.5 hrs</td>
</tr>
<tr>
<td>Ckt 5</td>
<td>1295</td>
<td>48 hrs</td>
</tr>
<tr>
<td>Ckt 6</td>
<td>3490</td>
<td>92 hrs</td>
</tr>
</tbody>
</table>

Now, let us analyze the performance of our tool in the context of fault-exclusion. Table 2 shows the results for the same circuits used in Table 1. Column 3 here shows the number of faults identified as untestable. Column 4 shows the time taken by our tool to analyze the fault-list. If we compare the time taken by our tool with the time taken by S-ATPG to analyze the fault list, we can see that our tool is much faster than S-ATPG. Moreover, unlike S-ATPG, we can be sure to exclude the transition faults identified by our tool as untestable from FTPG without losing on fault-coverage.

<table>
<thead>
<tr>
<th>Circuit</th>
<th># faults</th>
<th># untestable</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ckt 1</td>
<td>2042</td>
<td>10 min</td>
<td></td>
</tr>
<tr>
<td>Ckt 2</td>
<td>6108</td>
<td>55 min</td>
<td></td>
</tr>
<tr>
<td>Ckt 3</td>
<td>4384</td>
<td>50 min</td>
<td></td>
</tr>
<tr>
<td>Ckt 4</td>
<td>1453</td>
<td>6 min</td>
<td></td>
</tr>
<tr>
<td>Ckt 5</td>
<td>8466</td>
<td>2.5 hrs</td>
<td></td>
</tr>
<tr>
<td>Ckt 6</td>
<td>3782</td>
<td>30 hrs</td>
<td></td>
</tr>
</tbody>
</table>

Moreover, our tool can also be used to identify scan-based patterns that can cause potential yield loss. Functionally untestable transition faults can get detected during enhanced scan, broadside or skewed-load testing, and hence patterns that detect such faults can potentially cause a good chip to fail and result in yield loss. In order to avoid such yield loss, enhanced scan, broadside and skewed-load ATPGs can be guided to avoid generating patterns that incidentally detect the faults we identify as functionally untestable.
Table 3: Fault Coverage refinement for transition faults

<table>
<thead>
<tr>
<th>Circuit</th>
<th>ATPG FC</th>
<th>Refined-FC</th>
<th>% refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ckt 1</td>
<td>56.58</td>
<td>78.60</td>
<td>22.02</td>
</tr>
<tr>
<td>Ckt 2</td>
<td>58.87</td>
<td>86.12</td>
<td>27.25</td>
</tr>
<tr>
<td>Ckt 3</td>
<td>71.24</td>
<td>85.52</td>
<td>14.28</td>
</tr>
<tr>
<td>Ckt 4</td>
<td>74.20</td>
<td>86.05</td>
<td>11.85</td>
</tr>
<tr>
<td>Ckt 5</td>
<td>42.17</td>
<td>66.77</td>
<td>24.6</td>
</tr>
<tr>
<td>Ckt 6</td>
<td>31.48</td>
<td>46.30</td>
<td>14.82</td>
</tr>
<tr>
<td>Ckt 7</td>
<td>15.3</td>
<td>39.42</td>
<td>24.12</td>
</tr>
</tbody>
</table>

In the context of scan based testing, we view our tool as a potential performance booster for scan-based ATPG (in terms of savings in execution time and refinement of fault-coverage calculations). To understand the impact of the untestable transition faults identified by our tool on the performance of scan-based ATPG, consider the results reported in Table 3 and Table 5. Column 2 in Table 3 reports the fault-coverage (FC) achieved by scan-based ATPG if knowledge of untestable transition faults is not available. Here FC is simply the ratio of the detected faults to the total #faults. Column 3 shows the value of FC if the knowledge of untestable transition faults is also factored into the analysis. Here, FC = #detected faults / (#total faults - #untestable transition faults). Finally, column 4 shows the refinement in fault coverage metric achieved with the knowledge of untestable transition faults identified by our tool.

Now, it can be argued that similar FC refinement (using the same formula as used for transition faults) can be achieved even for stuck-at faults. So, we used an internal tool based on symbolic simulation and SAT to identify untestable stuck-at faults, and calculated the refinement achieved using the knowledge untestable stuck-at faults.

Table 4: Fault coverage refinement for stuck-at faults

<table>
<thead>
<tr>
<th>Circuit</th>
<th>% refinement in FC (for stuck-at faults)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ckt 1</td>
<td>0</td>
</tr>
<tr>
<td>Ckt 2</td>
<td>0.82</td>
</tr>
<tr>
<td>Ckt 3</td>
<td>0.06</td>
</tr>
<tr>
<td>Ckt 4</td>
<td>0.55</td>
</tr>
<tr>
<td>Ckt 5</td>
<td>0.87</td>
</tr>
<tr>
<td>Ckt 6</td>
<td>6.1</td>
</tr>
<tr>
<td>Ckt 7</td>
<td>0.52</td>
</tr>
</tbody>
</table>

Table 4 shows the results for untestable stuck-at faults. It can be seen the refinement in FC achieved using knowledge of untestable stuck-at faults is not as significant as the refinement achieved for transition faults, which means that the return-on-investment in identifying untestable transition faults can be greater than that for stuck-at faults.

Table 5 shows the time taken by ATPG to target the transition faults identified as untestable using our tool. Column 2 in Table 5 shows the time taken by ATPG to target transition faults identified as untestable with a backtrack limit of 20,000 while column 3 shows the time taken by ATPG with a backtrack limit of 100,000. In Table 5, results are shown only for the large circuits, since this is where the performance of ATPG is affected most due to the presence of untestable faults.

Table 5: Execution time of ATPG for untestable transition faults

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Time taken by ATPG for:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20,000 backtrack</td>
</tr>
<tr>
<td>Ckt 5</td>
<td>30 hrs</td>
</tr>
<tr>
<td>Ckt 6</td>
<td>1.5 hrs</td>
</tr>
<tr>
<td>Ckt 7</td>
<td>6 days</td>
</tr>
</tbody>
</table>

It can be observed that ATPG takes time in hours and days to analyze faults that our tool identifies as untestable, which indicates that by using our methodology, we are able to identify untestable faults which are hard to target for ATPG. Finally, we also show results that justify the need for a special tool such as ours that targets identification of untestable transition faults. Specifically, we show that the knowledge of untestable stuck-at faults is not enough to identify untestable transition faults.

Table 6: Need for untestable transition fault identification tool

<table>
<thead>
<tr>
<th>Circuit</th>
<th># Unt. with stuck @ data</th>
<th># Unt. using our tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ckt 1</td>
<td>0</td>
<td>2042</td>
</tr>
<tr>
<td>Ckt 2</td>
<td>140</td>
<td>6108</td>
</tr>
<tr>
<td>Ckt 3</td>
<td>25</td>
<td>4384</td>
</tr>
<tr>
<td>Ckt 4</td>
<td>122</td>
<td>1453</td>
</tr>
<tr>
<td>Ckt 5</td>
<td>400</td>
<td>8466</td>
</tr>
<tr>
<td>Ckt 6</td>
<td>1913</td>
<td>3782</td>
</tr>
</tbody>
</table>

Column 2 in Table 6 shows the # of transition faults identified as untestable using just untestable stuck-at fault data. If we compare the data in column 2 with the data in column 3 (number of untestable transition faults identified by our tool), we'll notice that the values in column 2 comprise a very small % of the values in column 3, indicating that knowledge of untestable stuck-at faults is not enough by itself to identify untestable transition faults.

9. Conclusion

In this paper, we presented a new technique that can be used to identify untestable transition faults in latch based designs with multiple clock domains, while highlighting some complexity issues associated with such designs with respect to untestable fault identification. Our tool does not assume any testing methodology (i.e., enhanced scan, broadcast or skew-load), and since transition faults identified as untestable using our tool are sequentially untestable, these faults can be used to guide test-pattern generation for transition faults in any testing methodology. Our tool is also capable of handling transition faults of different sizes, making the tool generic. We conducted experiments on several industrial circuits and discussed results that expressed the usefulness...
of our tool in the context of both functional test pattern generation and scan-based test pattern generation. We also justified the need for this work by showing that knowledge of untestable stuck-at faults is not enough to identify untestable transition faults.

REFERENCES