A Symbolic Technique for the Efficient ATPG of Speed-Independent Circuits

Marco A. Peña\textsuperscript{†}, Enric Pastor\textsuperscript{†} and Jordi Cortadella\textsuperscript{‡}

\textsuperscript{†} Department of Computer Architecture
Universitat Politècnica de Catalunya
08034 Barcelona, Spain
\{marcoa, enric\}@ac.upc.es

\textsuperscript{‡} Department of Software
Universitat Politècnica de Catalunya
08034 Barcelona, Spain
jordic@lsi.upc.es

Abstract

This paper presents a new symbolic ATPG approach for stuck-at faults in speed-independent asynchronous circuits. A closed system is built by composing the circuit model and the specification. Then, a fault is injected in the circuit and the closed system is analyzed by symbolic model checking techniques. The fault is manifested if a discrepancy state is reached. A sequence of events to reproduce the failure in a real circuit is generated during the test application phase. Such sequence is transformed into synchronous test vectors, suitable for commercial testers.

The main contribution of this work is the use of the circuit’s specification to efficiently drive the analysis, and the assumption of the fundamental mode of operation in order to reduce the state space required for the analysis and test pattern generation.

1 Introduction

Major research efforts in the field of asynchronous circuits have been devoted to solve the problems of automatic synthesis and verification of circuits [3]. However, testing is still to be satisfactorily solved.

The absence of global synchronization in asynchronous circuits makes controllability and observability significantly more costly than in synchronous ones. A circuit cannot be “single-stepped” through a sequence of states, therefore typical synchronous test techniques cannot be directly applied. Moreover, due to the large amount of feedbacks and state holding elements, design for testability requires a high area overhead [5]. Finally, asynchronous circuits may have hazards or races when faulty, and these delay faults are very difficult to test [6]. Conversely, some asynchronous circuits that use local handshakes to synchronize operations can be easier to test, since a stuck-at fault on a handshake signal causes a communication deadlock, an effect easy to detect.

Some classes of asynchronous circuits are self-checking under certain fault models, i.e. a fault halts the circuit while it is in normal operation. Speed-independent, delay-insensitive and quasi-delay-insensitive circuits are self-checking under the output stuck-at [2], the input stuck-at [5] and the isochronic transition [12] fault models, respectively. For asynchronous circuits designed under certain delay assumptions, the problem of ensuring some bounds for the delay along paths in a fabricated circuit has been considered [14, 7]. Design for testability techniques have been also proposed, such as the insertion of observation and control points [5] or test signals [12], as well as scan-path techniques [7, 1]. Finally, since commercial testers are inherently synchronous, it has been proposed to test asynchronous circuits by synchronous test vectors [1, 11].

In [1], the asynchronous circuit is modeled as a synchronous one by cutting feedback loops with virtual synchronous flip-flops, such that state-of-the-art synchronous ATPG techniques can be used. However, the obtained test vectors must be validated on the asynchronous circuit by means of simulation. Such validation has been shown to be optimistic [11], thus invalid, in the presence of non-confluence caused by a fault.

In [11], the non-faulty circuit is analyzed to find all input sequences that can be used synchronously, such that neither non-confluence nor oscillation is produced. Then, the behavior of the circuit is abstracted in terms of a synchronous deterministic finite state machine, which is analyzed as follows. As a first step, Random TPG is used to quickly cover a significant number of faults. With the remaining faults a symbolic 3-phase ATPG approach (fault activation, state justification and state differentiation) is used. Finally, each test sequence is efficiently simulated over other faults trying to find all those faults also covered by the same sequence.

The results obtained under the input stuck-at fault model, show a very high fault coverage for speed-independent circuits and promising coverage rates for hazard-free circuits with bounded delays. Moreover, since the specification is not used, a wide class of asynchronous circuits can be analyzed with this approach. However, the drawback is found in the 3-phase ATPG approach, being very complex and costly in terms of CPU and memory. This fact is specially noticeable in those circuits where a low fault coverage is achieved by the Random TPG step due to their topology.

This paper presents an ATPG technique to produce synchronous test vectors for speed-independent circuits. It might be applied after the Random TPG and prior to the “brute force” 3-phase ATPG, to overcome the complexity problems shown by the approach presented in [11]. The major features of the presented approach are:

- The specification is used to reduce the complexity of the state space exploration.
- The analysis is based on known efficient symbolic model checking techniques.
- Produced test vectors can be used with synchronous testers, thanks to the fundamental mode assumption.

The paper is organized as follows. In Section 2 the proposed ATPG methodology is outlined. Section 3 briefly introduces the formal models used along the paper. Section 4 introduces the fundamental mode of operation required for testing asynchronous circuits using synchronous testers, and discusses some preliminary results. Section 5 reviews the testability notion in such context. Section 6 illustrates the methodology with a simple example. Finally, Section 7 gives some conclusions and discusses future work directions.

2 Overview

The presented approach (see Figure 1) relies on the stuck-at fault model but, unlike in typical ATPG algorithms for synchronous circuits, the structure of the circuit is not used. The functional equivalence of the stuck-at faults is used instead.
Test pattern generation is driven by the circuit specification, by means of efficient symbolic model checking techniques for speed-independent circuits [10].

The approach starts by choosing a possible fault \( f \), which is injected in the circuit model. Then, the new model is confronted against the specification of the fault-free circuit. The detection of discrepancy states, is related to the presence of fault \( f \), which causes either a premature firing or an inhibited transition. The detection of such failures is subject to the observability at the primary outputs of the circuit, in a total stable state, i.e. state differentiation (see Section 5).

**Definition 2.1 (Failures)**

A signal transition is inhibited if it should occur according to the specification, but it does not occur.

A premature firing corresponds to a signal transition that occurs earlier than specified.

The detection of a failure is not useful if we cannot extract information about how to reproduce the discrepancy (fault activation). Hence, test patterns which detect the presence of such fault in the circuit under test, must be generated. A back-trace process performed from the discrepancy state to the initial one, generates a sequence of transitions reproducing the failure (state justification). The set of test vectors for fault \( f \) is directly derived from such sequence. The procedure continues by choosing a remaining fault and performing the above steps until all faults have been analyzed.

Speed-independent circuits are 100\% testable under the output stuck-at fault model [2], since faults cause the circuit to halt prematurely in a stable state. In our approach, the main analysis is performed using the fundamental mode of operation (Section 4). Therefore, all stable states are analyzed, and hence all output stuck-at faults are automatically covered. On the other hand, some input stuck-at faults can be left uncovered since their effect might not be observable in a stable state. However, since the approach is intended to be used in a general ATPG framework, like that of [11], the input-stuck at fault model is used, in order to cover as many faults as possible and to leave less work load for the costly 3-phase ATPG.

3 Models

**State Transition Diagrams (STDs)** [15] constitute the lowest level model for the representation of asynchronous circuits. STDs consist in an explicit enumeration of the state space of the circuit. An STD is a 5-tuple \( (S, E, A_I, A_O, \lambda) \), where \( S \) is a set of states, \( E \subseteq S \times S \) is a set of edges (state transitions), \( A_I \) is a set of input signals, \( A_O \) is a set of non-input signals, and \( \lambda : S \rightarrow \{0, 1\}^{I_{A_I} \cup A_O} \) is a total labeling function that encodes each state with a binary vector of signal values. Rising (falling) transitions of signal \( a \) are denoted \( a^+ \) (\( a^- \)), and generic transitions \( a \) (see the example in Figure 2 (b)). In every edge connecting a pair of states \( s_i \) and \( s_j \), \( \lambda(s_i) \) and \( \lambda(s_j) \) must differ exactly in one signal value, say signal \( a \). Then, signal \( a \) is said to be excited (opposite to stable) in \( s_i \). State \( s_j \) is directly reachable from state \( s_i \), if there is an edge from \( s_i \) to \( s_j \). More generally, for any pair of states \( s_i, s_j \in S \), \( s_i \) is reachable from \( s_j \) if there is a sequence of transitions leading from \( s_i \) to \( s_j \).

**3.1 Circuit Model**

In synchronous circuits the state depends on a subset of signals called state signals, including some input and flip-flop signals. The order of the transitions along combinational paths is not relevant. The only limitation is that they all must occur in a limited cycle time. Conversely, asynchronous circuits can be seen as an arbitrary interconnection of gates and delay elements. Therefore, states must be defined by the values of all signals, since feedback loops are not cut by clocked flip-flops.

Speed-independent circuits rely on the unbounded delay model [3], under which the next state of a circuit only depends on its present state. A gate is excited if its output
differs from the function it implements, and is stable otherwise. Provided the STD \( \langle S^C, E^C, A^C_1, A^C_2, \lambda^C \rangle \) capturing the underlying state space of a circuit, a next-state function \( \delta_C : S^C \rightarrow A^C_2 \rightarrow S^C \) can be defined for each gate implementing a non-input signal. Function \( \delta_C(s, a) \) returns either the state reached from \( s \) by switching signal \( a \) if it was excited, or \( s \) if it was stable.

### 3.2 Specification Model

**Signal Transition Graphs (STGs)** where introduced by [13, 4] as a specification formalism for asynchronous sequential circuits. An STG is an interpreted Petri net [8], and as such it is capable to explicitly capture the notions of causality, concurrency and choice.

A Petri net (PN) is a 4-tuple, \( N = \langle P, T, F, m_0 \rangle \), where \( P \) is a set of places, \( T \) is a set of transitions, \( F \subseteq (P \times T) \cup (T \times P) \) is a flow relation, and \( m_0 : P \rightarrow N \) is the initial marking. A marking of a PN is an assignment of a nonnegative integer to each place. The structure of a PN defines a set of firing rules that determine its behavior. Transition \( t \) is **enabled** when each predecessor place is marked. When \( t \) fires, one token is removed from each predecessor place and one token is added to each successor place. The PN moves from one marking to another by firing one of the enabled transitions.

An STG is a triple \( \langle N, A, \Lambda \rangle \), where \( N \) is a Petri net, \( A = A_1 \cup A_2 \) is a set of (input and non-input) signals, and \( \Lambda : T \rightarrow A \times \{1, -1, +, -\} \) is a transition labeling function. Transitions in an STG describe value changes at the signals of a circuit. A signal transition can be represented by \( a_i \cdot (or \ a_i \cdot)\) for the \( j \)-th rising (falling) transition of signal \( a_i \), while \( a_i \cdot \) is a generic name for either a rising or a falling transition. An STG can be directly translated into an STD \( \langle S^S, E^S, A^S_1, A^S_2, \lambda^S \rangle \) by transforming the reachable markings into states [13, 4]. Figure 2 shows an STG and its corresponding STD. Finally, a next-state function \( \delta_S : S^S \times T^S \rightarrow S^S \) can be defined, that transforms a marking (including signal values) by firing a transition.

The state space, the next-state functions of the circuit and the specification have been modeled in terms of boolean algebras and characteristic functions, allowing an efficient symbolic manipulation by means of BDDs (see [9]).

### 4 System Analysis

#### 4.1 Specification-Circuit Composition

The main part of the analysis is performed on a closed system composed by the circuit and its specification in terms of an STG. Note that an STG not only models the behavior of the circuit, but also the behavior of the environment in which the circuit will work. The closed system establishes a relationship between the interface signals of the circuit and its specification/environment, in a one-to-one fashion.

The idea of synchronization between the specification and the circuit is simple. It is assumed that the initial state of the STG is consistent with that of the circuit, and it is stable. The reachable states are calculated by means of symbolic breadth-first traversal techniques. The traversal starts by firing an input transition synchronically in both the specification and the circuit. Then, the STG and the circuit are left to stabilize separately according to the fundamental mode of operation. Therefore, output and internal signals of the circuit will switch independently of those of the STG. The synchronization is produced again by comparing the values of the interface signals in the total stable states reached. If a discrepancy is found the traversal stops. Then, a backwards analysis, like that used to produce failure traces in formal verification techniques as [10], is started to generate a sequence of transitions which reproduce the fault effect. If no discrepancy was found the process continues by firing a new input transition.

The states of the specification and the circuit are kept separately. The state of the specification is represented by a binary vector \( s_s \in S^s \), where each \( c_i \) in \( s_s = (c_1, \ldots, c_i, \ldots, c_n) \) is the value of signal \( a \) of the STG. In the same way, the state of the circuit is represented by a binary vector \( s_c \in S^c \), where each \( s_i \) in \( s_c = (s_1, \ldots, s_i, \ldots, s_n) \) is the value of signal \( s_i \) of the circuit. Synchronization between the specification and the circuit are produced when switching input signals and when checking the binary codes of the total stable states reached after the independent stabilization (Figure 3). Internal signals fire independently to each other. Premature firings are manifested by a discrepancy state in which some output signal in \( s_c \) switched, while its corresponding output signal in \( s_s \) did not. On the other hand, inhibited transitions are manifested by a discrepancy state in which an output signal in \( s_c \) did not switch, while its corresponding output signal in \( s_s \) did.

#### 4.2 Fundamental Mode

The asynchronous behavior of the circuits under test must be adapted to the synchronous operation mode of testers. The adaption can be done by imposing the **fundamental mode** of operation [3] during test generation. Fundamental mode assumes that the circuit starts in some stable state, i.e. all signals have fixed values and have no tendency to change. By definition, the circuit persists in a stable state until an input change from the environment occurs. After that, the environment is not allowed to change the inputs until the entire circuit stabilizes (a new stable state is reached).

**Definition 4.1 (Total Stable State)**

A state \( s \in S \) is a total stable state if only input transitions are enabled on it. That is, neither internal nor output transitions can fire from \( s \):

\[
\forall (s, s') \in E : \lambda(s)[a] \neq \lambda(s')[a] \Rightarrow a \in A_2 ].
\]

**Definition 4.2 (Input Successor State)**

Given a total stable state \( s \in S \) and a transition \( a \cdot \) of an input signal \( a \in A_1 \) excited on \( s \), the Input Successor State, \( ISS(s, a \cdot) = s' \), is the state reached from \( s \) by firing \( a \cdot \) :

\[
\{ (s, s') \in E \land \lambda(s)[a] \neq \lambda(s')[a] \land a \in A_1 \}.
\]

**Definition 4.3 (Output Burst of States)**

Given an input successor state \( s \in S \), the Output Burst of States of \( s \), \( OBS(s) \), is the maximal connected set of states reached iteratively from \( s \), by just firing any sequence of non-input transitions.
An approach for the state space exploration of a circuit, can be easily derived in terms of these definitions. Let us assume \( s_0 \in S \) be the initial state, which is a total stable state\(^2\). For this state, and for each input transition \( a \in A \), the corresponding OBS set is computed. Then, the corresponding OBS set is calculated for each input successor state (note that an OBS set could lead to more than one total stable state in cases of non-determinism, for example). The process repeats again starting from the total stable states in the computed OBS set. Thus, given a stable state, an input transition is fired. Then, the circuit is left to stabilize, and no new input transition is allowed until all possible internal and output transitions have happened. Figure 4 depicts the symbolic traversal algorithm according to the fundamental mode of operation, while figure 5 shows the resulting state space exploration for the example in Figure 2. Notice that the visited states are a subset of the full state space. This is due to two main factors derived from the fundamental mode assumptions [9]:

- Only total stable states are used as “source” for further explorations.
- Given a total stable state, only one input transition is allowed to fire at a time.

It is important to note that in a conventional traversal for the verification of the speed-independence, the full state space is analyzed [10]. Conversely, in our ATPG methodology, though based in similar principles, the check for discrepancy states is done only on total stable states. No check is done in the unstable states because they will not be observable during the test application phase. Moreover, since the intermediate unstable states are no longer needed, there is no need to keep them along the traversal. These two facts result in an important improvement in CPU and memory cost.

Tables 1 and 2 summarize some results about the fundamental mode traversal obtained with an experimental tool running on a SUN Ultra 170E workstation with 265Mb of main memory. Data in Table 1 is organized as follows. Columns ‘X’, ‘Y’ and ‘Z’ contain the total number of states, the number of states reached by the fundamental mode traversal algorithm, and the number of total stable states, respectively. Note that an important reduction in the number of visited states is achieved by the fundamental mode algorithm and that the reduction is even greater if only total stable states are taken. Thus, visited states and total stable states only represent the 27\% and the 7\% of the total number of states, respectively. Table 2 contains data for benchmarks with a huge state space, where those percentages are even better. Columns ‘A’, ‘B’ and ‘C’ contain respectively the number of states of the complete state space, the number of BDD nodes required to represent them, and the seconds of CPU employed. Similarly, columns ‘D’, ‘E’ and ‘F’ contain the number of total stable states, the number of BDD nodes, and the CPU seconds, if a fundamental mode analysis is done. Notice that some circuits could only be analyzed in the second case due to memory limitations. CPU time is slightly bigger although in cases like par_16 it is lower due to better variable reordering along the traversal.

5 Testability

Due to the synchronous nature of testers, only stable states can be observed during the test application phase. Thus the notion of fault detectability can be defined by:

**Definition 5.1 (Testable fault)**

A fault is testable in an asynchronous circuit if there exists a sequence of test vectors \( \tau = t_1, t_2, \ldots, t_n \) such that, all potential sequences of total stable states generated by \( \tau \) from the initial state \( s_0 \quad (s_0 \overset{t_1}{\rightarrow} s_1 \overset{t_2}{\rightarrow} s_2 \ldots s_m) \), end up into a total stable state \( s_m \) in which the failure can be observed at the primary outputs.
Many of the faults in asynchronous circuits introduce hazards, i.e. undesired switching activity not considered in the original specification of the circuit. Hazards are a potential source of circuit malfunction and —what is more relevant for the test purposes— non-determinism and oscillation [11]. A hazardous circuit may or may not deviate from its expected behavior depending on the working conditions and particular delays in the circuit. Figure 6 pretends to clarify this point for some particular cases. Shaded circles correspond to discrepancy states.

Figure 6 (a) shows an example in which a given fault cannot be tested with a test sequence $t_1$, $t_2$, $t_3$, $t_4$. In this example, after applying the test vector $t_1$ from state $s_1$ the circuit may non-deterministically, either evolve correctly through states $s_2$, $s_3$, and $s_4$, or may end up into a discrepancy state. Hence, it is impossible to ensure the fault detection because it can be done only sometimes. Figure 6 (b) shows a typical example of successful test in presence of hazards. Corresponding to the inherent non-deterministic nature of hazards, from state $s_1$ at every new test vector $t_2$ and $t_3$ the circuit may evolve into a discrepancy state or a correct one, eventually ending up into a discrepancy state after vector $t_4$. In the example of Figure 6 (c), the same test sequence may evolve into two different circuit behaviors. After applying vector $t_2$ the circuit evolves to discrepancy states through state $s_8$ in one case, and states $s_9$ and $s_{10}$ in the second case. In both examples (b) and (c) the fault can be successfully tested.

The main difference between asynchronous and synchronous circuits is that in the former, time is continuous, while in the latter time is discrete with the clock period being an indivisible unit of time. This period is chosen in such a way that it is longer than the critical path delay of the combinational logic. A similar reasoning can be done in order to set the clock period of the test machine when testing an asynchronous circuit. That is, the clock period must be set to the maximum time required for the circuit to stabilize given an input signal transition. This time is related to the length of the sequences of states between two total stable states, and strongly depends on the delays of the gates responsible of the state changes along such sequences [9, 11].

### Table 2: More experimental results about the traversal

<table>
<thead>
<tr>
<th>Circuit</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>pata5</td>
<td>658</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>pata6</td>
<td>1059</td>
<td>5</td>
<td>666</td>
</tr>
<tr>
<td>phil26</td>
<td>1059</td>
<td>5</td>
<td>666</td>
</tr>
<tr>
<td>phil27</td>
<td>1059</td>
<td>5</td>
<td>666</td>
</tr>
<tr>
<td>phil28</td>
<td>1059</td>
<td>5</td>
<td>666</td>
</tr>
</tbody>
</table>

### Table 6: Three situations of testability analysis

<table>
<thead>
<tr>
<th>Circuit</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc</td>
<td>21</td>
<td>21</td>
<td>15</td>
</tr>
<tr>
<td>atmel</td>
<td>24</td>
<td>23</td>
<td>15</td>
</tr>
<tr>
<td>ch150</td>
<td>26</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>ch172</td>
<td>12</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>diachi</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>ebergen</td>
<td>18</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>nak-pa</td>
<td>58</td>
<td>36</td>
<td>12</td>
</tr>
<tr>
<td>nowick</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>fc-cet</td>
<td>15</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>PMR</td>
<td>18856</td>
<td>3982</td>
<td>768</td>
</tr>
<tr>
<td>full</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>wxyz</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>half</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>hybrid8</td>
<td>80</td>
<td>56</td>
<td>20</td>
</tr>
<tr>
<td>tlc</td>
<td>28</td>
<td>28</td>
<td>14</td>
</tr>
<tr>
<td>sntp-wfd</td>
<td>22</td>
<td>22</td>
<td>8</td>
</tr>
<tr>
<td>IMR</td>
<td>21848</td>
<td>4200</td>
<td>768</td>
</tr>
<tr>
<td>input</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>irred</td>
<td>324</td>
<td>82</td>
<td>20</td>
</tr>
<tr>
<td>trimos-s</td>
<td>336</td>
<td>199</td>
<td>54</td>
</tr>
<tr>
<td>MR</td>
<td>8932</td>
<td>2520</td>
<td>768</td>
</tr>
<tr>
<td>SMR</td>
<td>2108</td>
<td>1092</td>
<td>333</td>
</tr>
<tr>
<td>men?2</td>
<td>52</td>
<td>26</td>
<td>10</td>
</tr>
</tbody>
</table>

Suppose we want to generate a test pattern for a stuck-at-0 fault on internal node $k$ of the circuit in Figure 2. The fault is injected in the circuit model which is then confronted against the specification. Provided the initial state, input transition $a+$ can fire. If the circuit is left to stabilize, output transitions $c+$, $d+$ and $e+$ can fire concurrently. Thus, total stable state $s_0 = (10111)$ is reached. In this state two input transitions may fire, namely $a+$ and $b+$. Let us assume that $b+$ fires. In such case, the specification expects output transition $c+$ to occur, but it will never happen in the faulty circuit, because the NOT-AND gate will never switch due to the stuck-at-0 on node $k$ (transition $c+$ is inhibited). In other words, according to the specification the circuit should have reached total stable state $s_{19} = (11011)$. Instead of this, it halts (i.e. stabilizes) in state $s_{14} = (11111)$. Using the subset of states visited along the traversal (Figure 7 (a)), a back-trace process is done from the expected final state $s_{19}$ to the initial state $s_1$. This process yields a sequence of signal transitions which reproduce the fault effect (see Figure 7 (b), where subsequences in brackets represent concurrent alternatives). If the unstable states are removed, the sequence of Figure 7 (c) is obtained, which can be used to calculate test vectors for checking the fault (see Figure 7 (d)).

As an example of premature firing detection, consider test pattern generation for a stuck-at-0 fault on the internal node $j$ of the same circuit. Again, transition $a+$ can fire from the initial state and the total stable state $s_0 = (10111)$ is reached after stabilizing the circuit. In this state, input transitions $a+$ and $b+$ can fire. If $b+$ fires, output transition $c+$ should occur, reaching total stable state $s_{15} = (11011)$ according to the specification. Instead of this, the faulty circuit stabilizes not only producing transition $c+$, but also transitions $d+$ and $e+$, and reaching state $(11000)$. That is, transitions $d+$ and $e+$ fire prematurely according to the specification, because $e+$ requires transition $a+$ to happen before, and $d+$ requires $b+$, but none of them have fired yet. Since the same state traversal than in previous example has been done before finding a discrepancy state, the same set of test vectors will allow the detection of the stuck-at-0 fault on internal node $j$. 
The paper has presented a novel efficient approach for the automatic generation of test patterns for speed-independent asynchronous circuits, under the input stuck-at fault model. The approach is based on a symbolic model checking strategy, and employs native asynchronous techniques along all its steps.

Since asynchronous circuits can only be stopped in stable states, the fundamental mode of operation is assumed along the analysis. Therefore, the produced test vectors can be used with current synchronous testers.

Unlike in other typical ATPG approaches, the structure of the circuit is not used. The functional equivalence of the stuck-at faults is used instead.

The circuit and its specification are represented by means of boolean functions and relations, allowing an efficient symbolic processing by using BDDs.

Test generation is driven by the circuit specification which, together with the fundamental mode assumption, provides an important complexity reduction. This has been shown by means of memory and CPU results, for normal size and huge examples. Since the targeted circuits are speed-independent, 100% output stuck-at fault coverage is guaranteed, event under the fundamental mode assumption. However, the input stuck-at fault model is used, in order to cover as many faults as possible and reduce the work load left to other costly but more exhaustive ATPG approaches. Therefore, the approach can be complementary used to overcome the complexity problems shown by some ATPG approaches for asynchronous circuits, like that in [11], thus, allowing the generation of test patterns for larger circuits in a reasonable amount of time.

Figure 7: Test pattern generation for a stuck-at-0 fault on node k of Figure 2 (c): (a) visited states, (b) sequence of transitions generated by the back-trace process, (c) after removing unstable states, and (d) sequence of test vectors

**References**


