Automatic Generation of Synchronous Test Patterns for Asynchronous Circuits

Oriol Roig Jordi Cortadella Marco A. Peña Enric Pastor
Department of Computer Architecture,
Universitat Politècnica de Catalunya.
Barcelona, Spain.

Abstract

This paper presents a novel approach for automatic test pattern generation of asynchronous circuits. The techniques used for this purpose assume that the circuit can only be exercised by applying synchronous test vectors, as is done by real-life testers.

The main contribution of the paper is the abstraction of the circuit's behavior as a synchronous finite state machine in such a way that similar techniques to those currently used for synchronous circuits can be safely applied for testing.

Currently, the fault model being used is the input stuck-at *model. Experimental results on different benchmarks show that our approach generates test vectors with high fault coverage.*

1 Introduction

Testing is one of the crucial problems that remains to be satisfactorily solved for asynchronous circuits. Since they are implemented as arbitrary interconnections of gates and their behavior is not subordinated to the timing dictated by a global clock, controllability and observability of internal signals become significantly more costly than in synchronous circuits [14]. Moreover, asynchronous circuits tend to have more feedbacks and more state holding elements than their synchronous counterparts. Thus, test pattern generation is harder and *design for testability* techniques like full *scan-path* may be unacceptably expensive. Furthermore, testers are inherently synchronous and cannot properly reproduce the environmental behavior for which an asynchronous circuit is designed.

Several studies have been presented in the last years addressing the testing and the design for testability of asynchronous circuits.

Some classes of asynchronous circuits are said to be *self-checking* under certain fault models, i.e. a fault will cause the circuit to halt while it is being operated normally. *Speed-independent, delay-insensitive* and *quasi-delay-insensitive* circuits are self-checking under the output stuck-at [3, 11], the input stuck-at [13] and the *isochronic transition* [21] fault models, respectively.

For asynchronous circuits designed under absolute delay assumptions, the problem of ensuring some bounds for the delay along each path must be considered. Under the *path delay fault* model [25], a given path in a fabricated circuit is faulty if it has a delay outside the specified interval. Several approaches tackle the testing of path delay faults in different ways [18, 17, 15].

Considerable effort has been devoted to proposing design for testability methodologies such as the insertion of *observation* and *control points* [13] or *test signals* [21], as well as *full scan-path* [17, 15, 2] and *partial scan-path* [16] techniques.

Since commercial testers are inherently synchronous, some authors have proposed testing asynchronous circuits by synchronous test vectors. In [5, 2], feedback loops are cut by virtual synchronous flip-flops during ATPG. Thus, ATPG can be done by using standard state-of-the-art synchronous techniques but the obtained test vectors must be validated on the asynchronous circuit. In section 6.1 the main differences with our approach will be discussed.

In this paper we propose a testing strategy for input stuck-at faults in asynchronous circuits with the following features:

- The circuit's behavior is modeled as a synchronous finite state machine. The original circuit specification is not required for testing.
- Test patterns are generated automatically by means of symbolic techniques.
- Test patterns can be synchronously applied to the inputs of the circuit and faults can be made observable at the outputs, thus allowing interaction with real-life synchronous testers.

The paper is organized as follows. In section 2 the overview and motivation of the method are outlined. Section 3 introduces the circuit model. Section 4 presents the synchronous abstraction of the asynchronous circuit. In section 5 the ATPG methodology is detailed. Finally, sections 6 and 7 respectively present results and conclusions.

2 Overview of the method

Unlike synchronous circuits, asynchronous circuits may manifest non-deterministic and/or unstable behavior if an inappropriate environment is applied to their inputs. This is a consequence of the usually contrived circuit topology, where many gates have reconvergent fanout. Two problems may arise if input patterns are not selected conveniently: *non-confluence of settling state* and *oscillation.* The former occurs when the final stable state of the circuit can

Work supported by ACiD-WG (Esprit 21949), CYCIT TIC 95-0419 and Departament d'Ensenyament de la Generalitat de Catalunya.

Design Automation Conference ®
Copyright © 1997 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all 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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.
0-89791-847-9/97/0006/$3.50 DAC 97 - 06/97 Anaheim, CA, USA

be different depending on the arrival times of the input events and the delays of the internal gates. This phenomenon can potentially lead to metastability [22]. The latter occurs when the circuit cannot rest in a stable state.

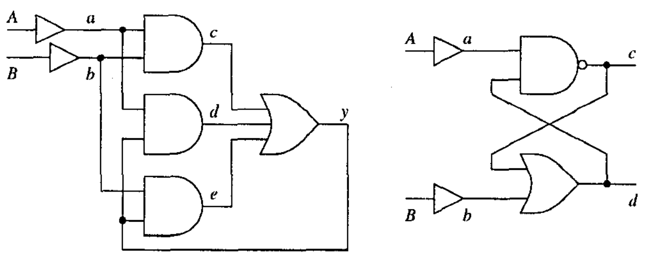

Figure 1: Circuits showing (a) non-confluence and (b) oscillation.

To show non-confluence of settling state we will take the circuit in figure 1(a). Let us assume the circuit is in the stable state $ABabcdey = 01010000$ and that the input pattern $AB = 10$ is applied. Even if A and B change simultaneously, because of the delay on primary inputs, we cannot assure that both a and b will change at the same time. A "competition" between all sensitized paths starts as soon as some input is switched. Two feasible sequences of gate transitions can be $a\uparrow, c\uparrow, b\downarrow, y\uparrow, d\uparrow, c\downarrow$ and $a\uparrow, c\uparrow, b\downarrow, c\downarrow$. If gate c is slow to fall the stable state 10101101 will be reached, otherwise the circuit will settle to state 10100000.

The other problem is oscillation or cycles of unstable states. Let the circuit in figure 1(b) be in the initial stable state $ABabcd = 000011$. If input A is set to 1, the circuit starts oscillating. After $a\uparrow$, the sequence of transitions $c\downarrow, d\downarrow, c\uparrow, d\uparrow$ is repeatedly generated and the circuit never stabilizes. Other circuits may present transient oscillations that should or should not be avoided depending on the maximum desired settle time.

There is a need of providing some technique that assures using only *valid* test vectors, i.e. input patterns that produce neither non-confluence of the settling state nor indefinite or too long oscillation cycles[1].

The overview of our testing approach is as follows.

1. The non-faulty circuit is analyzed to find all input sequences that can be used "a la synchronous", such that neither non-confluence nor oscillation is produced. After this analysis, the asynchronous circuit is modeled as a synchronous finite state machine (the *Confluent Stable State Graph, CSSG*, presented in section 4) with deterministic behavior.

2. *Random Test Pattern Generation* (Random TPG) on the *CSSG* is initially used to quickly cover a significant number of faults.

3. A symbolic ATPG strategy on the *CSSG* looks for a test sequence for each uncovered fault. Each test sequence is simulated on every remaining faulty circuit to find all other faults covered by the same sequence.

Both random TPG and fault simulation are efficiently performed by combining parallel and ternary simulation [24, 12]. Symbolic ATPG is performed by using BDD-based techniques similar to those used for synchronous finite state machines [10].

3 Circuit Model

An asynchronous circuit can be represented as an interconnection of *gates* and *delay elements*. A gate is a component with several inputs and outputs. At each gate output a function depending on the gate inputs is instantaneously computed. A delay is a single-input single-output element that reproduces its input after a certain time. *Wires* are used to interconnect gates and delays.

In our approach, asynchronous circuits are modeled following the *unbounded gate delay* model [19]. Under such a model, delay elements are attached only to gate outputs and the delay magnitude is positive and finite, but unknown. The delay type we assume is *inertial* delay, i.e. pulses shorter than the delay magnitude are filtered out. In the sequel we will refer to the pair formed by a gate and its associated delay simply as a *gate*.

Test vectors that would be valid under a bounded delay model, might be considered invalid under an unbounded delay model. On the other hand, test vectors generated assuming unbounded delays will also work on circuits with bounded delays. Therefore, our methodology while pessimistic, is independent of those aspects that may vary the gate delay, such as the technology, the fabrication process or the temperature at which the chips are being tested.

Each *primary input* of a circuit will be modeled as the input of a gate implementing the identity function. The circuit in figure 1(a) illustrates how primary inputs (A and B) are modeled. These buffers introduce the idea of delay associated with primary inputs.

3.1 Circuit State Graph (*CSG*)

In synchronous circuits the state depends on a subset of circuit signals called *state signals*. Usually this subset includes 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*. On the contrary, asynchronous circuits often have a more complicated structure. Since feedback loops are not cut by clocked flip-flops, the state of an asynchronous circuit is defined by all the binary values of both primary inputs and gates, rather than by a small subset of them.

A *state graph* (*SG*) is a pair $\langle \mathcal{S}, \mathcal{E} \rangle$, where $\mathcal{S}$ is the set of states and $\mathcal{E} \subseteq \mathcal{S} \times \mathcal{S}$ is the set of edges (or transitions).

A *circuit state graph* (*CSG*) is a 7-tuple $\langle \mathcal{S}, \mathcal{E}, \mathcal{P}, \mathcal{G}, S_0, \lambda_P, \lambda_G \rangle$, where $\langle \mathcal{S}, \mathcal{E} \rangle$ is a *SG*, $\mathcal{P} = \{p_1, \ldots, p_m\}$ is the set of primary inputs, $\mathcal{G} = \{g_1, \ldots, g_n\}$ is the set of gates, and $S_0 \subseteq \mathcal{S}$ is the set of initial states. The labeling functions $\lambda_P : \mathcal{S} \longrightarrow \{0,1\}^m$ and $\lambda_G : \mathcal{S} \longrightarrow \{0,1\}^n$ map each state s with a binary vector consisting of the values in s of primary inputs and gates, respectively.

Under the unbounded gate delay model the next state of a circuit uniquely depends on its present state. A gate is said to be *excited* if its output differs from the function it implements, and *stable* otherwise. If all the gates in a circuit are stable, the circuit is in a *stable state*. A *next state function* $\delta : \mathcal{S} \times \mathcal{G} \longrightarrow \mathcal{S}$ can be defined for each gate. Function $\delta(s, g_i)$ returns either the state reached by switching the output of g_i if it is excited or s if g_i is stable.

A *transition relation*, R, relates pairs of predecessor/successor states. If state s' is an immediate successor of state s, we say that both states are in relation R, denoted sRs' or $(s, s') \in R$.

By using the next state function of each gate, the transition relation associated to circuit gates can be defined as

$$R_\delta = \{(s, s') \in \mathcal{S} \times \mathcal{S} \mid (s \text{ is stable} \wedge s = s') \vee (\exists g_i \in \mathcal{G} \text{ such that } s' = \delta(s, g_i) \neq s)\} .$$

For each pair $(s, s') \in R_\delta$, if s is stable, its successor is the same s, otherwise the successor is obtained by switching an excited gate.

3.2 *CSG* in test mode

In our approach, asynchronous circuits are tested in synchronous mode: provided the circuit is stable, an input vector is applied and the circuit is allowed to, eventually, settle. The time between the

[1] In section 4 this notion of "too long" will be discussed.

application of two input patterns is called the *test cycle*. Until otherwise noted, we will assume the test cycle is *long enough* to let the circuit stabilize (unless it oscillates). Figure 2(a) illustrates a possible *CSG* in test mode. In principle, in a circuit with n inputs, the number of possible input patterns is 2^n, but in this picture only a few patterns are represented for the sake of simplicity. Labeled boxes represent stable states, while shaded circles are unstable states. The outgoing arcs from a stable state are labeled with the changes at the circuit primary inputs. Only in such arcs it is allowed more than one signal transition, whereas outgoing arcs from unstable states represent single signal transitions. The latter are not labeled for clarity. We will refer to this circuit state graph in test mode as *TCSG*.

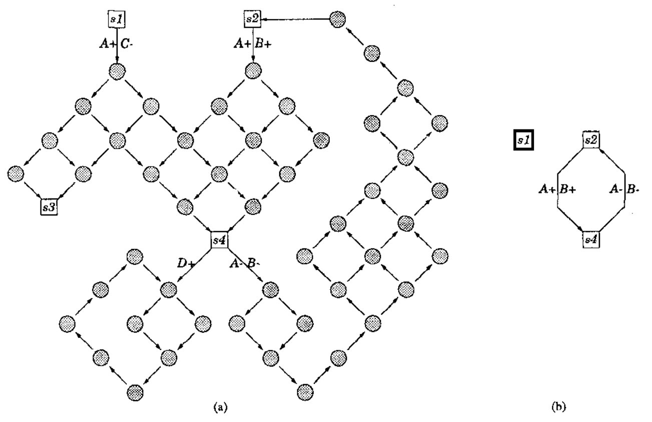

Figure 2: (a) A *TCSG* and (b) its corresponding *CSSG*.

The transition relation associated to input signals can therefore be defined as follows:

$$R_I = \{(s,s') \in \mathcal{S} \times \mathcal{S} \mid s \text{ is stable} \wedge \lambda_P(s) \neq \lambda_P(s') \;\wedge\; \lambda_G(s) = \lambda_G(s')\} \ .$$

Relation R_I describes all input patterns that can be applied to a stable state. Thus, sR_Is' if s is stable and s' only differs in certain number of inputs. This represents the situation in which several inputs have been changed but no gate has begun to switch yet.

The transition relation of a circuit in test mode is defined as $R = R_I \cup R_\delta$. Consequently, we can formally define a *TCSG* as a *CSG* such that $\mathcal{S}$ and $\mathcal{E}$ are strictly defined by the following recursion:

1. $S_0 \subseteq \mathcal{S}$.
2. $s \in \mathcal{S} \wedge sRs' \Rightarrow \begin{cases} s' \in \mathcal{S} \\ (s,s') \in \mathcal{E} \end{cases}$

The set $\mathcal{S}$ is the set of reachable states of a circuit in test mode, while $\mathcal{E}$ is the transition relation R restricted to $\mathcal{S}$. $\mathcal{S}$ can be calculated by using a symbolic traversal algorithms similar to the ones described in [10, 7].

4 Synchronous abstraction of the *TCSG*

This section explains how the *TCSG* is pruned in such a way that the input patterns that produce neither non-confluence nor oscillation are considered as valid candidates for test sequences. Roughly speaking, the *TCSG* will be reduced to a set of stable states and edges between stable states. For an edge (s, s') to exist, s must be stable and s' must be stable and the only state reached at the end of the test cycle. The finally obtained state graph will only contain the confluent and stable behavior of the original *TCSG*, hence the acronym *CSSG*, standing for *Confluent and Stable State Graph*.

We will use figure 2 as an example. Let us assume that $s1$ and $s2$ are initial states. A vector producing non-confluence is $A+C-$ applied to state $s1$, since either $s3$ or $s4$ can be nondeterministically reached. If vector $D+$ is applied to state $s4$ the circuit oscillates. The only valid vectors are $A+B+$ and $A-B-$ applied respectively to $s2$ and $s4$. This fact is manifested in figure 2(b). Note that the initial state $s1$ appears in the *CSSG*, even though no valid input pattern can be applied to it. Nevertheless, still some fault could be detected when forcing $s1$ as reset state.

4.1 Estimation of the test cycle

Unbounded gate delays and "long enough" test cycles are unrealistic assumptions for testing. Instead, bounded gate delays and short test cycles must be assumed. Moreover, the analysis of oscillation conditions is a difficult problem still under investigation.

Let σ be the longest sequence of transitions from a stable state s to the final stable state or states when certain input pattern is applied to s. If α is the longest gate delay, then $\tau = \alpha \cdot |\sigma|$ is an upper bound of the test cycle. On the contrary, if a test cycle of length t is desired, then $k = \lceil t/\alpha \rceil$ can be an estimation of the maximum number of allowed transitions before the circuit finally stabilizes. This is just an approximation we will use henceforth, but once the layout is provided, a more accurate cycle time can be calculated.

4.2 Practical computation of the *CSSG*

In order to calculate the *TCSG* synchronous abstraction, we first will define the pairs of states (s, s') such that s' is reached from s at the end of the test cycle. Each pair has an associated input pattern, given by the different values of inputs in s and s'. For the sake of clarity, subsequent pairs (s, s') will be assumed such that s is stable and s' is reached by propagating a single input pattern applied to s. We call the set of all these pairs of states the *test cycle relation*. For practical reasons we will assume that the circuit must settle in at most k transitions. The *k-step test cycle relation* (*TCR*^{k}) represents the pairs (s, s') distant at most k transitions. Formally, given a *TCSG* $\langle \mathcal{S}, \mathcal{E}, \mathcal{P}, \mathcal{G}, S_0, \lambda_P, \lambda_G \rangle$, *TCR*^{k} is defined as:

$$TCR^k = \{(s,s') \in \mathcal{S} \times \mathcal{S} \mid \exists s_1, \ldots, s_k \text{ such that } sR_Is_1 \;\wedge\; (\bigwedge_{i=2}^{k} s_{i-1}R_\delta s_i) \;\wedge\; s_k = s'\} \ .$$

The next step consist of removing invalid pairs of states. Vectors causing non-confluence are detected if pairs (s, s') and (s, s'') such that both s' and s'' have the same input values exist. Patterns producing oscillation or unacceptable long test cycle are found if s' is unstable. The *k-Confluent Stable State Graph*, denoted as *CSSG*^{k}, is formed by those pairs in *TCR*^{k} that present neither non-confluence nor cause the circuit to be unstable after k transitions. Formally, it can be defined as:

$$CSSG^k = \{(s,s') \in TCR^k \mid s' \text{ is stable} \wedge \nexists (s,s'') \in TCR^k \text{ such that } [s' \neq s'' \;\wedge\; \lambda_I(s') = \lambda_I(s'')]\} \ .$$

Informally the *CSSG*^{k} contains the following information. Each one of its nodes represents a stable state. An arc between two nodes s and s' exists if s' is stable and the only state reachable from s in at most k transitions by applying some input pattern.

5 Testing

Many techniques have been proposed for *Automatic Test Pattern Generation* (ATPG) for sequential synchronous circuits. As we have been explaining, however, non-confluence of settling state and oscillation make that those techniques cannot be directly applied to asynchronous circuits. Our approach resembles the *Three-phase ATPG* [8] proposed for synchronous circuits. We also propose a method with three phases: *fault activation*, *state justification* and *state differentiation*, described in sections 5.1 to 5.3. The way these three phases are implemented, though, will be different because of the asynchronous nature of circuits. Section 5.4 introduces Random TPG and fault simulation as techniques to increase the speed of the whole approach.

5.1 Fault activation

The first step to generate a test is to find a set of states that activate or excite the fault. It is easy to see that the fault signal x stuck-at-c is excited in some stable state if $x \neq c$. Since the set of stable reachable states has been already obtained during *CSSG* computation, finding the stable states exciting a fault is straightforward.

In most examples, there is always some stable state that excites a fault. However, it can occur that some signal always equals either 0 or 1 when the circuit is stable and only takes the opposite value in some unstable states. This situation arises when a signal switches an even number of times between stable states. Finding a test for such faults is left directly to the last phase, explained in section 5.3.

5.2 State justification

Justifying a state means to provide a sequence of input vectors that drive the circuit from the initial or reset state to that particular state. In our case, a sequence of test vectors that put the circuit in some of the excitation states must be given.

By using the reachability information it is easy to give a justification sequence. This sequence will put the correct circuit in a state that excites a given fault. However, the test vectors applied on the faulty circuit may result in a sequence of states that differs from that obtained in the correct one. In addition it has to be taken into account that some available sequences for the correct circuit can cause a faulty one to diverge or oscillate.

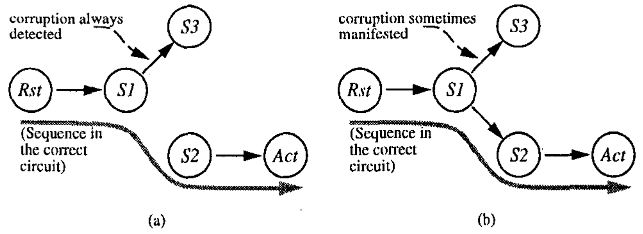
corruption always detected
S3
Rst
S1
(Sequence in the correct circuit)
S2
Act
(a)
corruption sometimes manifested
S3
Rst
S1
(Sequence in the correct circuit)
S2
Act
(b)

Figure 3: Corruption detected (a) always and (b) sometimes.

As noted in [8], there could be *corruption*, so the fault would manifest before. In a synchronous circuit the sequence that produces the corruption can always be taken as a new shorter excitation sequence. However, in an asynchronous circuit corruption has to be noticed in all terminal stable states. If this symptom does not appear in some stable states, the entire sequence has to be applied. The consequence when testing the real circuit will be that sometimes the fault will be detected before others.

Figure 3 illustrates this by means of an example. According to the reachability analysis done in the correct circuit, the sequence of states $Rst \rightarrow S1 \rightarrow S2 \rightarrow Act$ is exercised. Act indicates the proposed activation state actually being justified. When the same inputs are applied to a faulty circuit the following situations might be observed. The stable state $S1$ in figure 3(a) has $S3$ as its successor, instead of $S2$ as in the correct circuit. Because of the different behavior of the correct and the faulty circuit, the fault can be detected before expected. The case in figure 3(b) is different. Now depending on the delays of the gates, two states, $S2$ and $S3$, are reachable from $S1$. Since the fault can not always be detected, we have to apply the full sequence of input vectors. During the real test operation however, the fault may be detected before.

5.3 State differentiation

Once a fault has been excited, it still has to be made observable. The most favorable case occurs when the fault is propagated to some primary output. In general, the fault will propagate to some memory element. By applying successive input vectors we have to make the difference noticeable at a primary output. The *CSSG* gives all the feasible input vectors that can be used in each state. All of them are simulated by using similar techniques to the ones described in section 3.2, and the sequence resulting in a shorter test length is chosen.

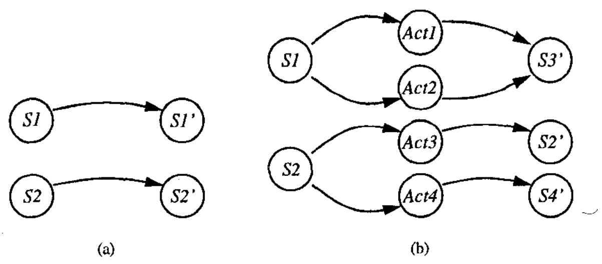

Figure 4: (a) Correct circuit (b) Faulty circuit.

As an example we can use figure 4. The different Act_i are the fault activating states, while $S1$ and $S2$ are their stable predecessors. In the correct circuit, $S1'$ and $S2'$ are reached, respectively, from $S1$ and $S2$. In the faulty circuit, $S3'$ is always reached from $S1$, therefore there is an appropriate excitation vector. However, depending on the gate delays, from $S2$ the fault can either be detected ($S2'$) or not ($S4'$). In the latter case, the test would not be conclusive.

5.4 Improving ATPG performance

The three phases described above are sufficient to find a test for any testable fault. However, they may be time consuming. Next we describe how techniques used to improve the speed of synchronous ATPG algorithms can be adapted to asynchronous ones.

Fault simulation is commonly used to find out if a test for a given fault also detects other faults. When a test is found to detect a fault, the same input patterns are simulated on the remaining faulty circuits. This technique will be efficient only if fast simulation algorithms are provided. Symbolic algorithms are good at managing multiple states of a same circuit. The problem when simulating a fault is just the opposite: dealing with a same state for each different faulty circuit. Then *parallel simulation* [24] is widely used because of its speed.

Non-confluence and oscillation are problems that have to be taken into account in simulation as well. A very efficient, though conservative, method called *ternary simulation* [12], has been widely used to detect when an input vector causes critical races. This conservativeness, however, does not affect the fault coverage of our approach. Let us assume that a given test detects some fault. The objective of fault simulation is just to find out if the same test detects other faults. If ternary simulation says that a test is unable to cover other faults, when in fact it could, tests for those faults can still be found by the previous three phases.

In ternary simulation a signal can have any of the three following values: 0, 1 or Φ. The symbols 0 and 1 have their usual boolean meaning, whereas the symbol Φ stands for an uncertain value which is neither 0 nor 1.

Ternary simulation consists of two algorithms namely A and B. Algorithm A sets each signal to the least upper bound of its current value and its evaluation. The result is that unstable signals are set to Φ. By repeating this process, uncertainties are propagated through the circuit. Algorithm B sets each signal to its evaluation. Consequently, some signals are set to a known value (either 0 or 1). Let us assume the circuit is in state s and we apply input vector a. After algorithms A and B we reach the final state s'. We can conclude that if all the signals in s' have a definite value (0 or 1), s' is the only successor of s when a is applied. On the contrary, if some signal in s' has an unknown value (Φ), either there are several final stable states or the circuit oscillates.

It has been proved that ternary simulation is polynomial in the number of circuit gates [6]. This is due to the fact that in the worst case $2n$ states are produced, n being the number of gates. In each state at most n function evaluations are required. Therefore, detection of critical races and/or oscillation can be detected in $O(n^2)$ for each pair of stable states and input pattern.

Random Test Pattern Generation has turned out to be a very efficient method in finding a test for an important number of faults at a very low CPU cost [4]. The number of faults covered by this technique highly depends on the circuit, but coverage ratios between 40% and 80% are commonly achieved. By using ternary simulation with Random TPG the speed of the overall approach can be improved in similar percentages.

Table 1: Experimental results (speed-independent)

	output-s		input-s					
example	tot	cov	tot	cov	rnd	3-ph	sim	CPU
alloc-outbound	32	32	66	66	51	12	3	52
atod	26	26	40	40	36	1	3	5
chu150	26	26	52	50	50	0	0	10
converta	22	22	44	44	20	23	1	8
dff	20	20	44	40	7	33	0	18
ebergen	32	32	70	70	32	38	0	43
hazard	20	20	44	44	22	22	0	7
master-read	62	62	130	130	55	75	0	5049
mmu	60	60	136	136	44	92	0	21067
mp-forward-pkt	28	28	58	58	57	1	0	6
mr1	60	60	140	139	5	134	0	27231
nak-pa	40	40	80	80	68	5	7	43
nowick	28	28	54	54	54	0	0	4
ram-read-sbuf	42	42	82	82	54	26	2	465
rcv-setup	20	20	36	36	31	5	0	4
rpdft	32	32	62	62	60	1	1	14
sbuf-ram-write	50	50	102	102	40	60	2	1760
sbuf-send-ctl	40	40	86	86	45	41	0	254
sbuf-send-pkt2	40	40	116	116	31	85	0	7137
seq4	40	40	86	84	28	52	4	256
trimos-send	58	58	132	132	6	126	0	16030
vbe10b	50	50	114	110	22	88	0	5534
vbe5b	22	22	42	41	33	8	0	9
vbe6a	38	38	80	78	17	61	0	562
Total FC	100.00%		99.16%					

6 Results

We show the effectiveness of our ATPG methodology over a set of benchmarks. Table 1 presents the results obtained for speed-independent and table 2 for hazard-free circuits with bounded delays. Both sets of benchmarks have been automatically synthesized from the same specifications, the former by Petrify [9] and the latter by SIS [23].

Results in the tables are structured as follows. The second and third columns respectively present the total ("tot") and covered ("cov") number of faults under the single output stuck-at fault model. The fourth and fifth columns show analogous results for the single input stuck-at fault model. The next three columns, namely "rnd", "3-ph" and "sim", detail the number of faults covered by each step of our approach. The last column reports the CPU time, in seconds, needed to find the whole set of test vectors. Benchmarks have been run on a Sun 4 workstation with a Sparc-20 processor and 64 megabytes of RAM.

Table 2: Experimental results (hazard free with bounded delays)

	output-s		input-s					
example	tot	cov	tot	cov	rnd	3-ph	sim	CPU
atod	44	44	66	66	52	11	3	487
chu150	26	26	48	46	44	2	0	15
converta	42	42	92	92	8	84	0	5008
ebergen	20	20	42	42	35	7	0	5
hazard	30	30	46	44	39	4	1	20
nowick	28	28	54	54	54	0	0	4
rpdft	36	36	48	48	48	0	0	8
trimos-send	72	24	126	29	12	17	0	254851
vbe10b	60	16	136	26	21	5	0	26774
vbe5b	32	32	52	52	42	10	0	19
vbe6a	62	18	126	29	23	6	0	29647
Total FC	69.91%		63.16%					

The input stuck-at fault model includes all output stuck-at faults. The results on output stuck-at faults are shown to illustrate that the well known theoretical result of speed-independent circuits being 100% output stuck-at fault testable in operation mode [3] still holds when our methodology is used.

Conversely, this is not true for the set of circuits generated by SIS. Most circuits present similar results to those of speed-independent circuits, but three benchmarks, *trimos-send, vbe10b* and *vbe6a*, presented a very poor fault coverage. This is due to the logic redundancies added by the synthesis tools in order to avoid spurious pulses in this type of circuits. Note that these examples also take a very long time to finish. When a test for an undetectable fault is searched, all possible input patterns are tried, thus time is wasted with no positive results. Finding out *a priori* undetectable faults may result in significant performance increase. In those cases with very low fault coverage, testability can be assisted by *partial scan-path* [16] or *variable phase splitting* [17].

The number of faults detected by random TPG depends highly on the example topology, but an average of 45% is achieved. This fact represents an important speed-up of our methodology. If a low coverage is achieved in the random step, much work will be left to the 3-phase step. 3-phase ATPG (fault activation, state justification and state differentiation) is the most complex step and the one dominating CPU time. Note that the highest test generation times correspond to those benchmarks where the random TPG step has covered a low number of faults (see e.g. *converta* and *trimos-send* in table 1). In some cases the same vector is reported to cover different faults. Due to the conservativeness of ternary simulation, it sometimes fails to detect equivalent tests. This is the reason for the low number of faults covered by fault simulation. Despite the the low number of faults covered by fault simulation, this last step is still performed because its execution time is negligible when compared to the 3-phase ATPG algorithm.

As a general consideration, such results can be significantly improved by speeding up the 3-phase step. Three possibilities we have in mind are: studying better variable ordering strategies in the use of BDDs, using hierarchical techniques similar to those utilized in some formal verification approaches [20] and classifying undetectable faults to avoid wasting time in covering them.

6.1 Discussion

Banerjee et al. [2] also propose synchronous testing of asynchronous circuits. They model the asynchronous circuit as a synchronous one by cutting feedback loops by virtual synchronous flip-flops. Hence, ATPG can be done by using efficient state-of-the-art synchronous techniques. Test vectors are validated afterwards on the asynchronous circuit by using zero-delay and unit-delay simulation [1]. Clearly, that validation can detect oscillation, but it is unable to identify non-confluence. This causes their approach to be optimistic.

Our approach assumes the pessimistic unbounded gate delay model, which assures that test vectors generated with our methodology are independent from the technology and the gate delays. Another difference between the method in [2] and ours is that we analyze the asynchronous circuit to find out those vectors that can be used in ATPG, so no further validation is needed. This analysis is done on the asynchronous circuit, rather than on a synchronous simplification. Consequently, our approach is computationally more expensive, but it can cope naturally with oscillation and non-confluence. Possibly, a hybrid method could take the best of both approaches.

7 Conclusions

Testing of asynchronous circuits is a problem still far from being solved satisfactorily. This paper has presented a method for ATPG based on well-known techniques for synchronous circuits.

The results shown in this paper indicate that asynchronous control circuits are highly testable without applying partial scan techniques. Automatic techniques to select those signals in which the insertion of scan paths can contribute to improve testability is also one of the goals to be pursued in the future.

The main contribution of this work is the synchronous abstraction performed over an asynchronous system such that real-life synchronous testers can be used to exercise the input signals.

This is only a preliminary work that will be further developed towards covering a wider spectrum of fault models (e.g. delay faults) with more efficient approaches. In the near future we want to explore the possibility of using hierarchy to tackle the testing of complex asynchronous systems. The synchronous abstraction of the circuit's behavior allows partitioning of large circuits into several interacting asynchronous circuits. We believe this feature will help to generate test patterns with techniques based on the composition of finite state machines.

REFERENCES

[1] S. Banerjee. Personal communication, Mar. 1997.

[2] S. Banerjee, S. T. Chakradhar, and R. K. Roy. Synchronous test generation model for asynchronous circuits. In *Proc. of the Int. Conf. on VLSI Design*, Bangalore, Jan. 1996.

[3] P. A. Beerel and T. H.-Y. Meng. Semi-modularity and testability of speed-independent circuits. *Integration, the VLSI journal*, 13(3):301–322, Sept. 1992.

[4] M. A. Breuer. A random and an algorithmic technique for fault detection test generation for sequential circuits. *IEEE Trans. on Comp.*, C-20(11):1364–1370, Nov. 1971.

[5] M. A. Breuer. The effects of races, delays, and delay faults on test generation. *IEEE Trans. on Comp.*, C-23(10), Oct. 1974.

[6] J. A. Brzozowski and C.-J. H. Seger. *Asynchronous Circuits*. Monographs in Computer Science. Springer-Verlag, 1995.

[7] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Trans. on CAD*, 13(4):401–424, 1994.

[8] H. Cho, G. D. Hachtel, and F. Somenzi. Fast sequential ATPG based on implicit state enumeration. In *Proc. Int. Test Conf.*, pages 67 - 74, Nashville, TN, Oct. 1991.

[9] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Congreso de Diseño de Circuitos Integrados*, Barcelona, Nov. 1996.

[10] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In *Proc. IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, Nov. 1989.

[11] I. David, R. Ginosar, and M. Yoeli. Self-timed is self-diagnostic. *Journal of Electronic Testing:Theory and Applications*, (6):219–228, Jan. 1995.

[12] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM J. Res. and Dev.*, 9:90–99, Mar. 1965.

[13] P. J. Hazewindus. *Testing Delay-Insensitive Circuits*. PhD thesis, California Institute of Technology, 1992.

[14] H. Hulgaard, S. M. Burns, and G. Borriello. Testing asynchronous ciruits: A survey. *Integration, the VLSI journal*, 19(3):111–131, Nov. 1995.

[15] K. Keutzer, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis for testability techniques for asynchronous circuits. *IEEE Trans. on CAD*, 11(1):87–101, Dec. 1995.

[16] A. Khoche and E. Brunvand. Testing self-timed circuits using partial scan. In *Proc. of the 2nd Working Conf. on Asynchronous Design Methodologies*, pages 160–169, London, May 1995.

[17] L. Lavagno, M. Kishinevsky, and A. Lioy. Testing redundant asynchronous circuits. In *Proc. EURO-DAC*. IEEE Computer Society Press, Sept. 1994.

[18] C. J. Lin and S. M. Reddy. On delay fault testing in logic circuits. *IEEE Trans. on CAD*, 6(5), Sept. 1987.

[19] D. Muller and W. Bartky. A Theory of Asynchronous Circuits. In *Annals of Computing Laboratory of Hardward University*, pages 204–243, 1959.

[20] O. Roig, J. Cortadella, and E. Pastor. Hierarchical gate-level verification of speed-independent circuits. In *Proc. of the 2nd Working Conf. on Asynchronous Design Methodologies*, pages 128–137, London, May 1995.

[21] M. Roncken and R. Saeijs. Linear test times for delay-insensitive circuits: a compilation strategy. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 13–27. Elsevier Science Publishers, 1993.

[22] C. L. Seitz. System timing. In *Introduction to VLSI Systems*, chapter 7. Mead & Conway, Addison-Wesley, 1980.

[23] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuits synthesis. Technical Report M92/41, UCB/ERL, May 1992.

[24] S. Seshu. On an improved diagnosis program. *IEEE Trans. on Electronic Comp.*, EC-12(2):76–79, Feb. 1965.

[25] G. L. Smith. A model for delay faults based on paths. In *Proc. Int. Test Conf.*, pages 324–349, Sept. 1985.