Computing and Informatics, Vol. 29, 2010, 1025–1043

DEDUCTIVE FAULT SIMULATION TECHNIQUE FOR ASYNCHRONOUS CIRCUITS

Roland Dobai, Elena Gramatová

Institute of Informatics
Slovak Academy of Sciences
Dúbravská cesta 9, 845 07 Bratislava, Slovakia
e-mail: {`roland.dobai, elena.gramatova`}`@savba.sk`

Manuscript received 13 October 2009; revised 21 April 2010
Communicated by Hana Kubátová

Abstract. Fault simulator for asynchronous sequential circuits (ASCs) needs to deal with hazards, oscillations and races. The simplest algorithm for simulating faults is the serial fault simulation technique which was successfully used for the ASCs. Faster fault simulation techniques, for example deductive fault simulation, was previously used for the combinational and synchronous sequential circuits only. In this paper a deductive fault simulator for the stuck-at faults of speed-independent (SI) ASCs is presented. An algorithm for the propagation of the fault lists is proposed which can deal with the complex gates of the ASCs. The implemented deductive fault simulator was tested using SI benchmark circuits. The experimental results show significant reduction of the computation time and negligible increase of the memory requirements in comparison with the serial fault simulation technique.

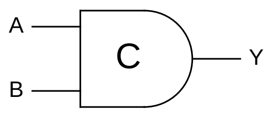
Keywords: Asynchronous circuits, testing, serial fault simulation, complex gates, deductive fault simulation, stuck-at faults

1 INTRODUCTION

Most of the digital sequential circuits designed today are synchronous. All their components share a common and discrete notion of time, as defined by a clock signal distributed throughout the circuit. The counterparts of the synchronous sequential circuits are the asynchronous sequential circuits (ASCs). They use *handshaking* between their components in order to perform the necessary synchronization and sequencing of operations [1]. Handshaking is a controlled, periodic exchange of synchronizing pulses between a digital transmitter and a receiver [2]. The transmitters and the receivers are the inner components of the ASCs.

This difference gives inherent properties to the ASCs that can be exploited to achieve lower power consumption, higher operating speed, less electro-magnetic noise emission, better modularity and better robustness in comparison with synchronous sequential circuits [1].

C-elements are used in the ASCs as basic construction elements. A C-element is a state-holding element and its symbol and truth table are shown in Figure 1. If inputs A and B of the C-element are set to logic zeros (ones) then the output Y is also set to logic zero (one). The output does not change with application of other combinations of input values and it holds the previous output value Y_{prev} [1].



A	B	Y
0	0	0
0	1	Y_{prev}
1	0	Y_{prev}
1	1	1

Fig. 1. Symbol and truth table of C-element

At the gate level, ASCs can be classified as self-timed (ST), speed-independent (SI), delay-insensitive (DI) or quasi-delay-insensitive (QDI) circuits. SI circuits operate correctly assuming positive finite delays in gates and ideal zero-delay wires. A circuit that operates correctly with positive finite delays in wires as well as in gates is a DI circuit. DI circuits with isochronous wire forks are called QDI circuits. A wire fork is isochronous if the signal transitions occur at the same time at all its end-points [1].

In this work only the SI circuits are considered but a tool for the SI circuits can be extended to handle other classes too. DI and QDI circuits can be represented under the SI timing model. Therefore, the original circuit can be handled by simulating the transformed circuit [3].

The SI circuits can be synthetized by the Petrify software tool [4] from signal transition graph (STG) specification. STG specification is used for modeling behavior of SI circuits. The Petrify tool can synthetize SI circuits with complex gates, with generalized C-elements, or even can perform the technology mapping. The complex gates are used in the ASCs for elimination of some hazards [1].

A hazard is an unwanted glitch on a signal and it is related to the dynamic operation of a circuit which relates to the input signals dynamics as well as to delays on the gates and wires of the circuit [1]. The hazards can be static or dynamic. The signal value with static-zero (static-one) hazard means to remain stable at logic zero (one) momentarily becomes logic one (zero) instead. If a signal value makes three or more transitions instead of a single transition ($0 \to 1$, $1 \to 0$), it is the dynamic hazard [5].

Several multi-valued logics have been proposed for hazard detection. The most complete is the 13-valued logic which is constructed as the product $A_3 \times A_5 \times A_3$, where A_3 is 3-valued logic and A_5 is 5-valued logic. The possible values of A_3 are logic zero 0, logic one 1 and do-not care X. The possible values of A_5 include the values of A_3 and two signal transitions $\uparrow$ and $\downarrow$. The logics are described as 3-tuples according to 13-valued algebra where the 1^{st} item of the 3-tuple represents the initiatory value, the 2^{nd} item the temporary intermediate value and the 3^{rd} item the final value. The items of the 13-valued logic are stable logic zero $(0, 0, 0)$, stable logic one $(1, 1, 1)$, do-not care (X, X, X), rising transtion $(0, \uparrow, 1)$, falling transition $(1, \downarrow, 0)$, static-zero hazard $(0, X, 0)$, static-one hazard $(1, X, 1)$, dynamic $0 \rightarrow 1$ hazard $(0, X, 1)$, dynamic $1 \rightarrow 0$ hazard $(1, X, 0)$, stabilizing one $(X, X, 1)$, stabilizing zero $(X, X, 0)$, destabilizing one $(1, X, X)$ and destabilizing zero $(0, X, X)$ [6, 7].

Another problem in the ASCs are the races. A race can exist in a circuit if two or more feedback signals are changing simultaneously. The race is critical if the order of changes can affect the final circuit state. Therefore another problem at the ASCs design is to avoid critical races [8].

A *defect* in an electronic system is the unintended difference between the implemented hardware and its intended design. A representation of a defect at the abstracted function level is called a *fault*. The term *fault* is used to refer to electrical, Boolean, or functional malfunctions [9, 10]. The faults of ASCs can be *process faults*, *transient faults*, or *delay faults*. Process faults originate from fabrication. Transient faults are those that might occur at some time, but are not stable in the sense they might not occur at other times. If a fault causes a circuit to exceed its timing specifications, but does not affect its logical function, it is said to be a delay fault [14]. The most known and used process fault model is the stuck-at fault model. The stuck-at fault is modeled by assigning a fixed logic zero or one to a signal line in the circuit [9, 10]. In this work only stuck-at faults are considered.

The fault simulator determines the fault coverage of a given set of input vectors for a given fault model or a given fault list by classification of the given target faults in the circuit as detected or undetected. The fault coverage is defined as the ratio of the number of detected faults to that of faults in the initial fault list for simulation expressed in percentage [9].

The serial fault simulation is the simplest algorithm for simulating faults. It is based on the following steps [9]:

1. The fault-free response of the circuit to the set of test vectors is saved to a file.
2. The circuit is simulated for each individual injected fault for the same set of test vectors used for fault-free simulation. The response of the circuit is compared with the fault-free response.
3. If the response is different then the fault is detected.
4. Steps 2–3 are repeated for all of the faults.
5. The fault coverage is computed based on the number of undetected faults and faults involved in fault simulation.

Fault dropping technique considerably speeds up the serial fault simulation. Fault dropping is the practice in which faults detected by a test vector are deleted from the fault list prior to the simulation of any subsequent test vector [11].

The fault simulation time can be further reduced by the deductive or the concurrent fault simulation technique. The main advantage of these fault simulation techniques is creation of detectable fault lists and their propagation to primary outputs (POs) for a current test vector in a single simulation overpass. The disadvantage of this technique is in higher memory requirements. All signal values in each faulty circuit are deduced from the fault-free circuit values and the circuit structure. Since the circuit structure is the same for all faulty circuits, all deductions are carried out simultaneously. The simulation proceeds by simulating a vector in the fault-free mode. Before simulating the next vector, a deductive procedure is applied to all lines in a level-order from primary inputs (PIs) to POs. In this process, fault lists are generated for each signal. The fault list of a signal is derived from fault lists at the inputs of the gate producing that signal and any faults associated with that gate. This process is called *fault propagation*. The fault list of a signal at any time during simulation contains the names of all faults in the circuit that can change the state of that line [9, 12, 13].

The fault propagation rules through the simple logic gates *OR* and *AND* are in Table 1 where I_1 is the set of gate inputs with logic one, I_0 is the set of gate inputs with logic zero, $j = \{0, 1, \ldots, n-1\}$ is the indexed input of the n-input gate and L_j is the fault list for input j. The propagation rule for the logic gate is selected based on the condition that the controlling set is empty. The controlling set is the set of inputs with the controlling value. In the case of logic operation *AND* the controlling value (non-controlling value) is the logic zero (one) and for the operation *OR* the logic one (zero).

AND gate		OR gate	
condition	propagation rule	condition	propagation rule
$I_0 = \emptyset$	$\left\{\bigcup\limits_{j\in I_1} L_j\right\}$	$I_1 = \emptyset$	$\left\{\bigcup\limits_{j\in I_0} L_j\right\}$
$I_0 \neq \emptyset$	$\left\{\bigcap\limits_{j\in I_0} L_j\right\} - \left\{\bigcup\limits_{j\in I_1} L_j\right\}$	$I_1 \neq \emptyset$	$\left\{\bigcap\limits_{j\in I_1} L_j\right\} - \left\{\bigcup\limits_{j\in I_0} L_j\right\}$

Table 1. Fault list propagation through simple gates

When it comes to commercial use of the ASCs the problem of test comes to attention. The testing methods for synchronous sequential circuits can not be applied directly to the ASCs. Design for testability (DfT) techniques and testing methods are in advanced stage for the synchronous circuits but for the ASCs there are still problems to be solved – lack of automatic test equipment (ATE), test generation algorithms, built-in self-test (BIST) mechanisms and DfT standards. These problems cause limited practical usability of the ASCs.

The rest of the paper is organized as follows. Section 2 contains related work to the ASCs testing. Section 3 presents the developed deductive fault simulation technique for the SI ASCs. Section 4 analyzes the achieved results and Section 5 concludes the paper with analyzing the results and future work.

2 RELATED WORK

The logic and fault simulator for the SI circuits (SPIN-SIM) has been developed and published [7, 15]. This simulator cooperates with the automatic test pattern generator (ATPG) named SPIN-TEST [16, 17]. SPIN-TEST generates tests for the SI circuits using the stuck-at fault model. SPIN-SIM adopts a 13-valued logic [6] to improve the hazard detection accuracy and maintains the relative order of causal signal transitions using time stamps. The time stamp is simple and only includes a signal group ID and a time. The group ID is used to indicate causal transitions; signal transitions with a causal relation are assigned the same group ID. The relative order of the causal transitions is recorded in the time field, which is incremented as the transition propagates. The 13-valued logic was successfully used also for the test generation of the delay faults of sequential circuits [18]. SPIN-SIM transforms a SI circuit to the combinational logic by replacing each C-element with a set of simple gates and cutting the feedback paths into pseudo-primary inputs (PPIs) and pseudo-primary outputs (PPOs). The reason of this transformation is to create a pure combinational representation of the ASC for which an external ATPG for combinational circuits can be used. The set of gates for representation of the function of the C-element is assembled to be hazard-free. These gates are so called pseudo-gates because they have zero-delays. SPIN-TEST uses ATPG ATALANTA [19] for generating tests over the stuck-at fault model. The A* search algorithm is used in SPIN-TEST which is a graph-search algorithm to find the least-cost path between two nodes of the graph and for the cost estimation a heuristic function is used. In SPIN-TEST the A* algorithm is applied to find the best sequence of test vectors to reach each of the previously generated test vector (one node of the graph) from the pre-specified initial state of the circuit (another node of the graph) [20]. SPIN-TEST uses as heuristic function the estimation of the hardness of the fault activation and propagation to the POs. This function is used by the A* algorithm to make the decision which test vectors to use. The serial fault simulation technique is implemented in the SPIN-SIM fault simulator.

Another serial fault simulator called Fsimac was also developed for the ASCs. Fsimac is also realized over the 13-valued logic and can be used only for the ASCs with bounded gate delays because it is based on the min-max timing analysis [23]. It can not be used for the SI circuits because they assume unbounded gate delays and therefore the maximum delay of the interconnected gates can not be computed.

For the deductive and concurrent fault simulation technique the rules for fault propagation through simple logic gates are well known [9] as well as through logic gates with arbitrary number of inputs [11], or through combinational blocks described by the function definition language (FDL), implemented in the LAMP system [21]. A deductive fault simulator was developed also for the synchronous sequential circuits where the fault lists for the sequential blocks are modeled by Moore-type automata and are propagated using the automata states [22].

3 DEDUCTIVE FAULT SIMULATION

This section describes the proposed deductive fault simulation technique for SI ASCs. The deductive fault simulation is based mainly on the fault list propagation algorithm. The main contribution of the paper is the new fault list propagation algorithm which is universal and can be used for any gate represented by a Boolean function expressed in disjunctive normal form (DNF). The developed deductive fault simulator can propagate the fault lists in the SI circuit with complex gates. The gate level fault list propagation algorithm is described in more detail in Subsection 3.1.

The deductive fault simulation technique was previously used mostly for combinational circuits and occasionally for synchronous sequential circuits. The proposed fault simulator is based on the adopted transformation principle of the ASCs to the combinational representation [7, 15]. The fault list from the PPO is propagated to the PPI if the fault list on the PPI is different than that on the PPO and the maximum number of iterations has not been reached. The developed deductive fault simulator uses the 13-valued logic [6] and the time stamps [7] for hazard detection. The time stamps are also dealing with the races. The oscillations are detected using a simple counter. The hazard detection, the handling of races and oscillations with the 13-valued logic and time stamps are adopted [7], therefore they are not described in the paper. The proposed deductive fault simulator considers faults as detected only on the hazard-free POs, just like the other fault simulators for ASCs. The fault simulation consists of only one simulation overpass during which the fault-free and faulty simulations are performed simultaneously. The fault-free simulation uses the 13-valued logic and detects the hazards in the circuit. The simulation of faulty circuits deals with the fault lists only and uses the (3-valued) logic value of the fault-free simulation to create the propagation rules. The 3-valued logic is given as the 3rd item of the ordered 3-tuple representation of 13-valued logic designating the value to which it is intended to stabilize in time. Therefore, for lines with hazards the final stabilized value is considered during the fault propagation. In such cases the fault propagation is not interrupted because it is not possible to determine at that point whether the hazard will be manifested on an output or not. The developed deductive fault simulation for the ASCs is described in detail in Subsection 3.2.

3.1 Gate Level Fault List Propagation Algorithm

The input Boolean function for the proposed algorithm must be in the DNF which is the disjunction of conjunctions. For example function $Y = (A \wedge B) \vee (\neg A \wedge \neg B \wedge C)$ is in the DNF, where $A \wedge B$ and $\neg A \wedge \neg B \wedge C$ are the conjunctions (logic AND operations) over the components and where the disjunction (logic OR operation) of these conjunctions is the Boolean function in DNF. For example $\neg A$, $\neg B$ and C are the components for the conjunction $\neg A \wedge \neg B \wedge C$. In the developed algorithm the symbol $A - B$ is used to denote a set difference and the symbol $A \cap B$ to denote an intersection between two sets A and B. The symbol & is used for the logic operation *AND*.

The meaning of the variables used in the algorithm is as follows:

logicValueForAnd – the Boolean value of the examined conjunction;

***andIntersectionList* (*orIntersectionList*)** – the list of faults for the examined conjunction (the whole disjunction) which contains the faults of inputs (of conjunctions) with a logic zero (logic one) which is the controlling value for the logic operation *AND* (*OR*);

***andUnionList* (*orUnionList*)** – the list of faults for the examined conjunction (the whole disjunction) which contains the faults of inputs (of conjunctions) with a logic one (logic zero) which is the non-controlling value for the logic operation *AND* (*OR*);

***andControllingSetIsEmpty* (*orControllingSetIsEmpty*)** – the Boolean value to indicate whether there is an input (a conjunction) with a logic zero (logic one) or not;

tmpFaultList – temporary fault list used at the assignments.

The listed variables are supplemented step-by-step with the examination of the complex gate represented by the Boolean function. The variables *andControllingSetIsEmpty* and *orControllingSetIsEmpty* are used to make the decision the formula from which row of Table 1 should be used for the fault propagation.

In the algorithm the following auxiliary functions are used:

portOf(k) – It returns the gate input of component k. For example if $k = \neg A$ then *portOf(k)* $= A$.

logicValueOf(p) – It returns the logic value of the gate input p.

faultListOf(p) – It returns the detectable fault list of the gate input p.

The proposed method for the gate level fault list propagation is described by the algorithm showed in Figure 2. At the beginning of the algorithm some initialization has to be made. The temporary lists are deleted and the variable called *orControllingSetIsEmpty* is set to true because it is indicating that none of the conjunctions of the whole Boolean function yet has logic one (which is the controlling value in the case of the logic operation *OR*). After that follows the analysis of the Boolean function which is described in more detail later. During this analysis the *orUnionList* and *orIntersectionList* are filled with the appropriate faults. This process is followed by determination of the final fault list based on the value of *orControllingSetIsEmpty*. The final fault list is determined by the formulas from the last column of Table 1.

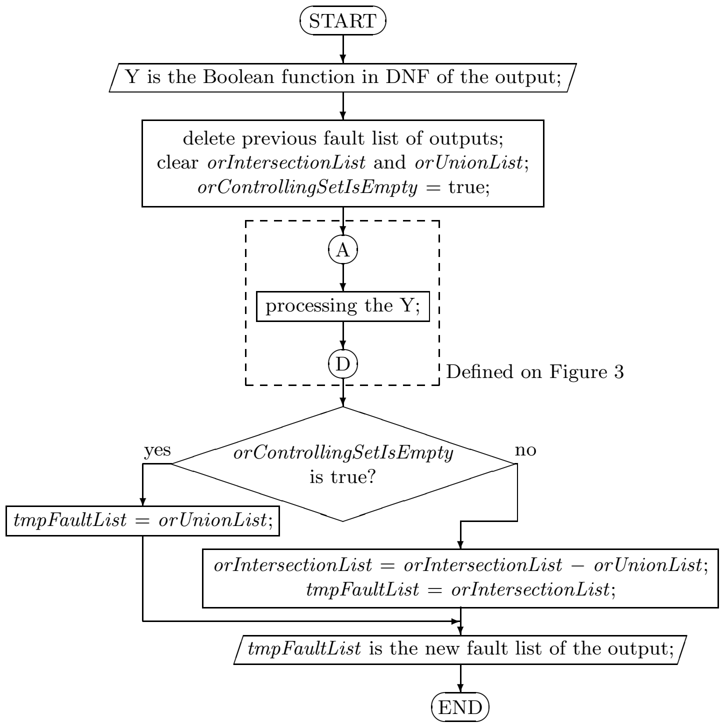

START
Y is the Boolean function in DNF of the output;
delete previous fault list of outputs;
clear *orIntersectionList* and *orUnionList*;
orControllingSetIsEmpty = true;
A
processing the Y;
D
Defined on Figure 3
orControllingSetIsEmpty is true?
yes
no
tmpFaultList = *orUnionList*;
orIntersectionList = *orIntersectionList* − *orUnionList*;
tmpFaultList = *orIntersectionList*;
tmpFaultList is the new fault list of the output;
END

Fig. 2. Gate level fault propagation algorithm

The process of analysis of the Boolean function which is used in Figure 2 is presented in more detail in Figure 3. We note all of the variables in the developed algorithm are global which means they are accessible from the sub-processes and after modification in the sub-process the results are accessible from the calling process. The analysis of the Boolean function consists of a loop which is used to evaluate each conjunction step-by-step. First some initialization has to be made for each conjunction. After that follows the analysis of the current conjunction which is described in more detail later. During this analysis the *andUnionList* and *andIntersectionList* are filled with the appropriate faults. This process is followed by determination of the fault list for the conjunction based on the value of *andControllingSetIsEmpty*. The fault list is determined by the formulas from the second column of Table 1. The faults from this list are put into the *orUnionList* or *orIntersectionList* based on the logic value of the examined conjunction. After that the analysis of the next conjunction begins.

The process of analysis of the conjunction which is used in Figure 3 is presented in more detail in Figure 4. This sub-process consists of a loop which is used to evaluate each component of the current conjunction. Firstly the logic value of the conjunction is determined in sequence. After that the faults of the components

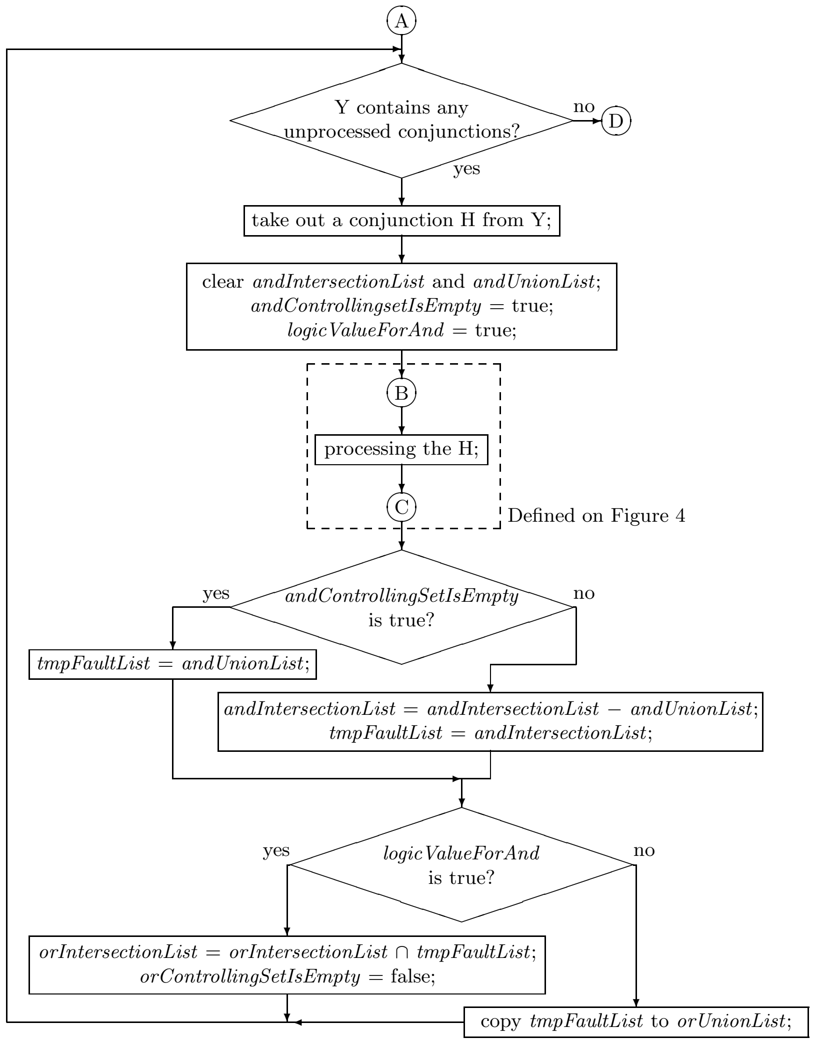

A
Y contains any unprocessed conjunctions?
no
D
yes
take out a conjunction H from Y;
clear andIntersectionList and andUnionList;
andControllingsetIsEmpty = true;
logicValueForAnd = true;
B
processing the H;
C
Defined on Figure 4
yes
andControllingSetIsEmpty is true?
no
tmpFaultList = andUnionList;
andIntersectionList = andIntersectionList − andUnionList;
tmpFaultList = andIntersectionList;
yes
logicValueForAnd is true?
no
orIntersectionList = orIntersectionList ∩ tmpFaultList;
orControllingSetIsEmpty = false;
copy tmpFaultList to orUnionList;

Fig. 3. Algorithm for analysis of the disjunction

gate input are put into *andIntersectionList* or *andUnionList* if the logic value of the current component is the controlling value for the logic operation *AND* or the non-controlling value, respectively.

3.1.1 Example

Consider the complex gate as a part of a circuit with the function $Y = (A \wedge B) \vee (\neg A \wedge \neg B \wedge C)$, where A, B, C are the inputs and Y is the output of the gate.

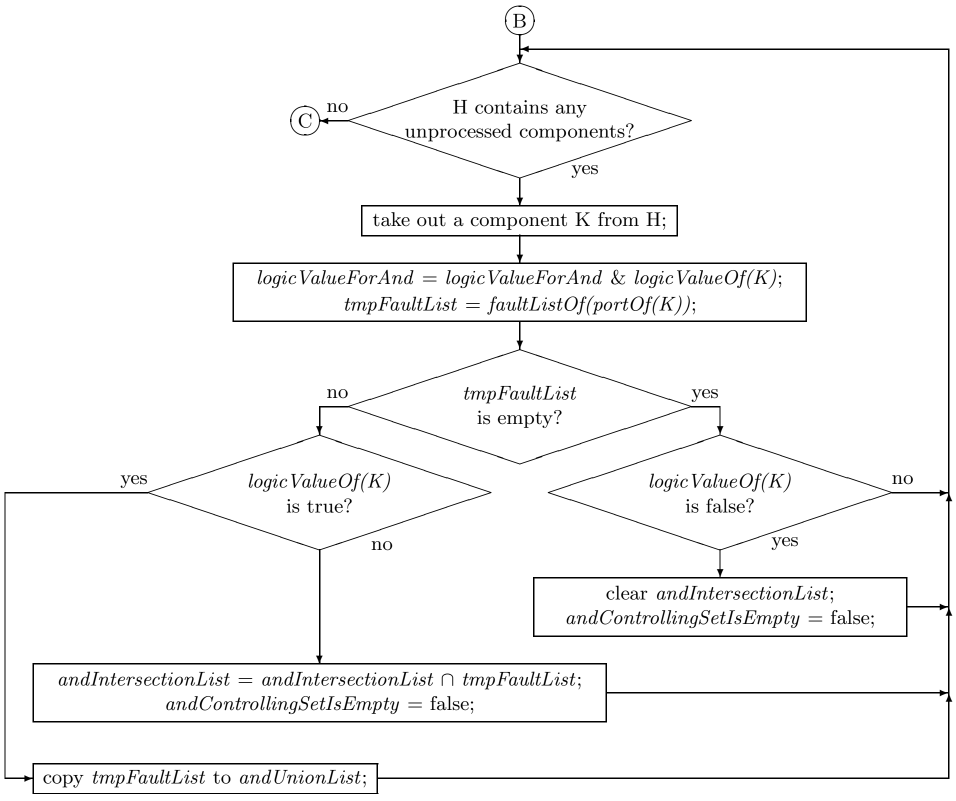

B
C
no
H contains any unprocessed components?
yes
take out a component K from H;
logicValueForAnd = logicValueForAnd & logicValueOf(K);
tmpFaultList = faultListOf(portOf(K));
no
tmpFaultList is empty?
yes
yes
logicValueOf(K) is true?
no
logicValueOf(K) is false?
no
yes
clear andIntersectionList; andControllingSetIsEmpty = false;
andIntersectionList = andIntersectionList ∩ tmpFaultList; andControllingSetIsEmpty = false;
copy tmpFaultList to andUnionList;

Fig. 4. Algorithm for analysis of a conjunction

Let us assume all of the inputs are set to logic one, i.e. $A = B = C = 1$, next G, H, L are lines in the circuit which are situated before the examined complex gate. The fault list on the input A is $L_A = \{A\_sa0, G\_sa0, H\_sa1\}$, on the input B is $L_B = \{B\_sa0, G\_sa0, H\_sa1\}$, and on the input C is $L_C = \{C\_sa0, H\_sa1, L\_sa1\}$. The label of faults consists of two parts. The first part defines the fault place and the second part identifies it as stuck-at zero ($sa0$) or stuck-at one ($sa1$) fault. The example is shown in Figure 5 where the function of the complex gate is represented with imaginary single gates and interconnections between them.

The proposed algorithm processes the function Y as follows:

- Analysis of the conjunction $A \wedge B$ – represented by the logic gate *and_1* in Figure 5.

  1. $A = 1$ which is the non-controlling value of the logic operation *AND* so the faults from L_A are put into the *andUnionList*.
  2. $B = 1$, so similarly the faults from L_B go into *andUnionList*. After this step $andUnionList = \{A\_sa0, G\_sa0, H\_sa1, B\_sa0\}$.

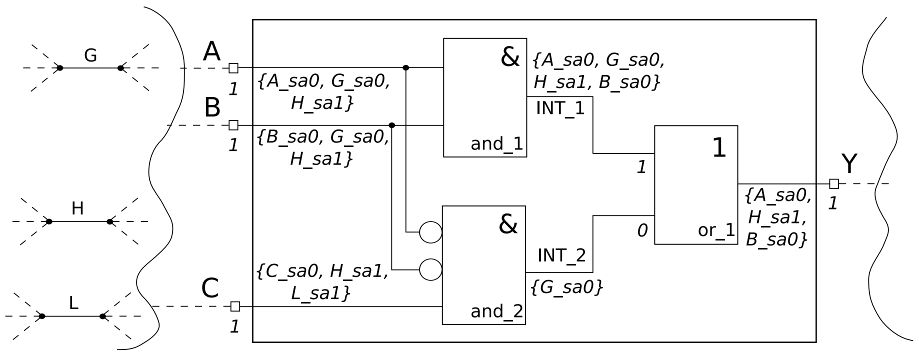


Fig. 5. Fault propagation through an example complex gate

3. *andControllingSetIsEmpty* = *true* because there are no inputs in this conjuction with the controlling value of logic operation *AND*. The faults from *andUnionList* go to *orIntersectionList* because $A \wedge B = 1$ which is the controlling value of the logic operation *OR*. The corresponding assignment in Figure 5 is made by propagation of the fault list $\{A\_sa0, G\_sa0, H\_sa1, B\_sa0\}$ to the internal interconnection *INT_1*.

- Analysis of the conjunction $\neg A \wedge \neg B \wedge C$ – represented by the logic gate *and_2* in Figure 5.

1. $\neg A = 0$ which is the controlling value of the logic operation *AND* so the faults from L_A are put into the *andIntersectionList*.
2. $\neg B = 0$, so $andIntersectionList \cap L_B$ will be the new *andIntersectionList*, i.e. $\{A\_sa0, G\_sa0, H\_sa1\} \cap \{B\_sa0, G\_sa0, H\_sa1\} = \{G\_sa0, H\_sa1\}$.
3. $C = 1$ which is the non-controlling value of the logic operation *AND* so the faults from L_C are put into the *andUnionList*.
4. *andControllingSetIsEmpty* = *false* so the fault list for this conjuction is *andIntersectionList* − *andUnionList* = $\{G\_sa0, H\_sa1\} - \{C\_sa0, H\_sa1, L\_sa1\} = \{G\_sa0\}$ and it goes to *orUnionList* because $\neg A \wedge \neg B \wedge C = 0$ which is the non-controlling value of the logic operation *OR*. The corresponding assignment in Figure 5 is made by propagation of the fault list $\{G\_sa0\}$ to the internal interconnection *INT_2*.

- Analysis of the disjunction – represented by the logic gate *or_1* in Figure 5.

1. *orControllingSetIsEmpty* = *false* so the fault list for the disjunction is *orIntersectionList* − *orUnionList*, i.e. $\{A\_sa0, G\_sa0, H\_sa1, B\_sa0\} - \{G\_sa0\} = \{A\_sa0, H\_sa1, B\_sa0\}$ which is the final fault list for the complex gate. The corresponding assignment in Figure 5 is made by propagation of the fault list $\{A\_sa0, H\_sa1, B\_sa0\}$ through the gate *or_1* to the output *Y*.

3.2 Global Fault Propagation Algorithm

The global fault propagation algorithm is based mainly on fault simulation technique of synchronous sequential circuits. The main difference is that not only one simulation overpass is performed through the combinational part of the circuit. These overpasses are repeated until the logic values and fault lists are not the same on each PPI-PPO pairs (a pair is developed by cutting a feedback path) or a maximum number of overpasses is not reached. The second difference consists in use of 13-valued logic and time stamps during the fault simulation.

The global fault propagation algorithm is shown in Figure 6 where *fbCounter* is used for counting the number of propagations through feedbacks, *MaxPass* is the maximum number of allowed propagations through feedbacks. The fault lists for the POs are deducted for all of the input vectors. At the beginning the input values are assigned to the PIs and the current time stamps are set. The time stamps are set only for those PIs where a signal transition is created by the current test vector. The time is measured from the beginning of the fault simulation and for the new test vector the incremented end simulation time from the previous test vector simulation will be used. The reason is to avoid the time consuming fault list removal at the beginning of the simulation. The new values will overwrite the existing ones in the circuit without causing any problems. During this first step the fault lists of PIs are erased only. If the PI has a fan-out the local stuck-at fault is assigned to this port based on the current logic value of this port (for a line with logic one a stuck-at zero and for a line with logic zero a stuck-at one fault is assigned – this principle is used later too). The reason is to universally handle the faults. We name the fault for a single line after the destination port and only in the presence of fan-out a fault named after the source port is inserted to the fault list (because in this case this fault is not equivalent with those at the destination ports). This principle is used later too.

After that all of the gates connected to the PIs are added to the list of gates for evaluation. The fault lists are moved from the PIs to the gate input ports together with the logic values and time stamps during this procedure. The local stuck-at faults based on the current logic values are also added to the input ports of the gates. After that an infinite loop is beginning its execution. This loop will be broken only if the list of the gates for evaluation becomes empty or the maximum number of iterations is reached. In this loop the gates from the list are evaluated. During this evaluation the logic values and the time stamps for the output ports are computed, the fault list is generated based on the presented algorithm in Subsection 3.1 and the time stamp is incremented. If the output port of the gate has a fan-out, then the local stuck-at fault based on the current logic value is assigned to the fault list at that output port, similarly to the assigment at the PIs. After the evaluation the logic values, the time stamps and the fault lists are moved to the ports of the connected gates, POs or PPOs. If the destination is a port of a non-pseudo-gate the local stuck-at fault based on the current logic value is added to the destination ports list (we ignore the pseudo-gates during the assigment of new faults because they only

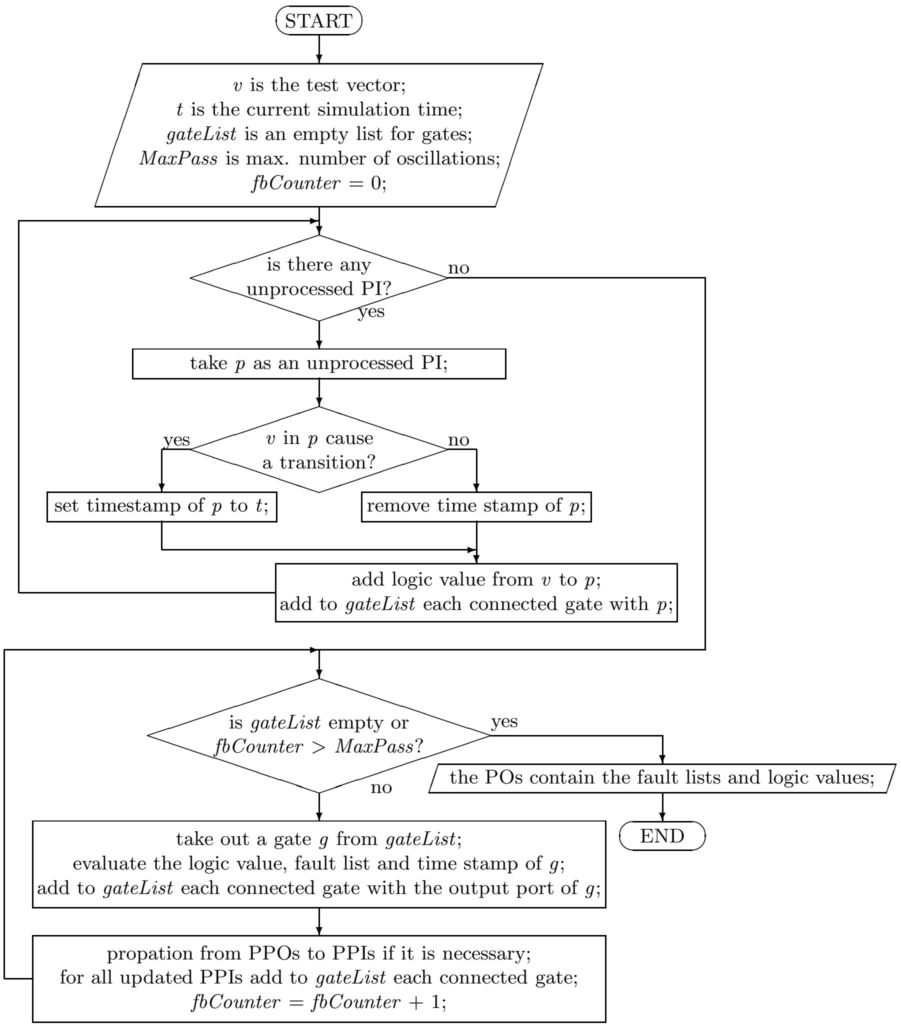

START
v is the test vector;
t is the current simulation time;
gateList is an empty list for gates;
MaxPass is max. number of oscillations;
fbCounter = 0;
is there any unprocessed PI?
no
yes
take p as an unprocessed PI;
v in p cause a transition?
yes
no
set timestamp of p to t;
remove time stamp of p;
add logic value from v to p;
add to *gateList* each connected gate with p;
is *gateList* empty or *fbCounter* > *MaxPass*?
yes
no
the POs contain the fault lists and logic values;
END
take out a gate g from *gateList*;
evaluate the logic value, fault list and time stamp of g;
add to *gateList* each connected gate with the output port of g;
propation from PPOs to PPIs if it is necessary;
for all updated PPIs add to *gateList* each connected gate;
fbCounter = *fbCounter* + 1;

Fig. 6. Global fault propagation algorithm

represent the memory element in the circuit and the new lines are not present in the ASC just in the combinational representation). All of these interconnected gates are added to list for evaluation. After the evaluation of each gate from the list the next iteration cycle will begin. Before that the logic values, time stamps and the fault lists are moved from the PPOs to the PPIs if the current logic values or the fault lists are different. The connected gates of those PPIs are inserted to the gate list whose logic values or fault list has been changed. After that the iteration counter is incremented and the next loop of the infinite loop begins. After the infinite loop has been broken the logic values and the fault lists at the POs are reported for the current test vector.

The list of the gates for evaluation is a queue. The gates are always placed to the end of the queue and are removed from the front of the queue. Therefore the gates are evaluated in the breadth-first way.

The faults are represented as single numbers in the deductive fault simulator. The other alternative could be the representation as strings which is "human-readable" but the work with faults in this form is computationally inefficient. Also the fault list is implemented not as a vector but as a set of ordered numbers.

In the proposed algorithm the wires of the circuit are represented as interconnections between the input and output ports of the gates. The stuck-at faults are assigned to these ports. As a part of the parsing process of the description of the circuit a unique even number is assigned to those ports where a potential stuck-at fault could be present. This is done because during the simulation process the fault lists are cleared frequently and new faults are added to them. The unique numbers make this process more effective. The fault identificator is created as a logic addition of the stuck-at fault type to the unique number. Therefore each stuck-at-zero fault will have an even identificator and each stuck-at-one fault an odd identificator.

Operations (such as copying, intersection, union, difference and comparison) are defined on the fault lists to make possible the execution of the algorithm from Subsection 3.1. Effectiveness of these operations is essential for global performance of the algorithm because they are executed relatively many times. Thanks to the given representation of faults and fault lists these operations can be executed in efficient way. For example the comparison could be performed as a comparison number-by-number and with a single overpass over the set. As it was examined and measured without these improvements the deductive fault simulator could not be faster than the serial fault simulator. The fault lists are deleted often and after that new faults are assigned to them. It was also found out that the performance is better this way. The checking which faults should remain in the list and which should be removed adds a significant overhead to the computation time.

4 RESULTS

For the simulation purposes a single random test pattern generator for the ASCs [17] was implemented. The implementation has been made in C++ using only the standard template library (STL). Evaluation has been done over a set of SI benchmark circuits synthetized by Petrify [4]. The STG specifications of these circuits are available on the Myers Research Group website [24].

The experiments were performed on a desktop computer (with AMD Athlon 64 X2 Dual Core 4400+ processor and with 2 GB of RAM) running Linux 2.6.24.

A simple utility was implemented for the experimentation purposes. The task of this utility is to measure the CPU time and the memory requirements of the implemented simulators. To measure the time it uses the GNU version of the *time* utility [25]. The *time* does not provide information about the memory requirements under Linux and is not reliable under FreeBSD (as found out during the implementation). Our implemented utility measures the requirements using the process information pseudo-file system (*/proc*) of the Linux kernel. The utility starts the fault simulator with a second's delay to make enough time for preparation of the measurement. During the measurement the resident memory sizes of the process are recorded and after the fault simulation the maximum value is reported.

The coverages of the stuck-at faults published in [17] and [7] are reported in the third and fourth columns of Table 2. The results of our serial and deductive fault simulator are in the fifth and sixth columns of the table. These fault coverages were achieved with 200 randomly generated test vectors while the results in [7] were achieved with 10 000 vectors. The number of test vectors was determined as the lower boundary without the decrease of fault coverage. These circuits were synthetized from the STG specifications [24] and manually checked and corrected the initial state of some circuits. As can be observed in Table 2, the fault coverages for our serial and deductive simulator are the same for each examined SI benchmark. The achieved fault coverage in the worst case is 94.12%.

Circuit name	Number of faults	Average fault coverage			
		[17]	[7]	serial	deductive
alloc_outbound	58	92 %	100.0 %	100.00 %	100.00 %
chu133	60	97 %	96.9 %	98.33 %	98.33 %
chu150	40	82 %	97.1 %	95.00 %	95.00 %
converta	56	46 %	91.9 %	96.43 %	96.43 %
dff	34	79 %	85.7 %	100.00 %	100.00 %
ebergen	46	N/A	95.7 %	100.00 %	100.00 %
half	34	N/A	100.0 %	94.12 %	94.12 %
hazard	40	86 %	97.0 %	100.00 %	100.00 %
master_read	132	46 %	97.7 %	95.45 %	95.45 %
mp_forward_pkt	66	95 %	100.0 %	100.00 %	100.00 %
nak_pa	76	91 %	100.0 %	100.00 %	100.00 %
nowick	50	98 %	100.0 %	100.00 %	100.00 %
ram_read_sbuf	84	89 %	100.0 %	100.00 %	100.00 %
rcv_setup	36	93 %	100.0 %	100.00 %	100.00 %
rpdft	26	92 %	100.0 %	100.00 %	100.00 %
sbuf_ram_write	82	78 %	100.0 %	100.00 %	100.00 %
sbuf_send_ctl	66	49 %	94.9 %	98.48 %	98.48 %

Table 2. Fault coverage comparison

Table 3 shows the CPU time and the memory requirements of the implemented serial and deductive fault simulators. The CPU times of the serial simulator are in the second column and the CPU times of the deductive one in the third column of Table 3. In most of the cases the deductive simulator is by 60% – 80% faster than the serial one. In some cases (for circuits *chu133*, *converta* and *half*) the deductive simulator is slower than the serial one. The reason rests in the fault dropping technique of the serial simulator. If fault coverage is achieved relatively close to 100% with a small amount of test vectors right in the beginning of the test, then the serial simulator will have to simulate only a small amount of faulty circuits to the end of the test because of the implemented fault dropping technique. If this fault coverage will not achieve 100% before the end of the test (so the fault simulation will not be terminated sooner), then the deductive simulator will have to propagate the fault lists (work with all faults) for the rest of the test. This will result in higher computation time for the deductive simulator. This problem could be eliminated by help of deterministic test pattern generator.

Circuit name	Time [s]		Time	Memory [kB]		Mem.
	ser.	ded.	decr.	ser.	ded.	incr.
alloc_outbound	0.19	0.07	63 %	1 668	1 700	2 %
chu133	0.23	0.29	−26 %	1 680	1 780	6 %
chu150	0.09	0.07	22 %	1 660	1 552	−7 %
converta	0.18	0.26	−44 %	1 560	1 744	12 %
dff	0.05	0.00	100 %	1 548	1 672	8 %
ebergen	0.07	0.01	86 %	1 552	1 700	10 %
half	0.08	0.13	−63 %	1 548	1 680	9 %
hazard	0.05	0.01	80 %	1 552	1 692	9 %
master_read	0.70	0.39	44 %	1 732	1 964	13 %
mp_forward_pkt	0.24	0.04	83 %	1 584	1 608	2 %
nak_pa	0.32	0.08	75 %	1 684	1 612	−4 %
nowick	0.14	0.05	64 %	1 664	1 704	2 %
ram_read_sbuf	0.25	0.06	76 %	1 688	1 756	4 %
rcv_setup	0.07	0.02	71 %	1 660	1 664	0 %
rpdft	0.04	0.00	100 %	1 536	1 640	14 %
sbuf_ram_write	0.15	0.01	93 %	1 688	1 748	2 %
sbuf_send_ctl	0.19	0.10	47 %	1 684	1 728	3 %

Table 3. Time and memory comparison

The memory requirements of the serial simulator are in the fifth column and the memory requirements of the deductive one in the sixth column of Table 3. The results are not accurate in some cases (for circuit *chu150* and *nak_pa*) because of low execution time of the fault simulation. In these cases lower requirements are reported for the deductive fault simulator than for the serial one. Another reason could be that for a small circuit the fault lists are also small which results in an even tinier difference in memory requirements; but even for the longer simulations the reported requirements are not more than 14%.

5 CONCLUSIONS

The deductive fault simulator was proposed for the SI ASCs and its implementation was tested over a series of SI benchmark circuits. The experimental results show

60 %–80 % reduction of the computation time and only max. 14 % increase of the memory requirements.

The developed fault simulator could not only work with the circuits synthetized by Petrify but also with SI circuits of any synthetizer. In that case only the input parser should be extended to be able to parse different circuit description format.

The simulator can be extended to handle other classes of the ASCs too because they can be represented under the SI timing model [3].

Some improvements of the test generation procedure are planned to be made. The random test pattern generator will be replaced with a deterministic generator to reduce the number of the test patterns and the simulation time. This improvement will also cause that the deductive simulator will be faster than the serial one for any circuit. Finally the fault simulator will be extended to handle not only stuck-at faults but also delay faults.

Acknowledgement

This project was supported by the grant VEGA 2/0135/08.

REFERENCES

[1] Sparsø, J.—Furber, S.: Principles of Asynchronous Circuit Design: A Systems Perspective. Kluwer Academic Publishers, Boston/Dordrecht/London, 2001.

[2] Gibilisco, S.: The Illustrated Dictionary of Electronics. McGraw-Hill, 2001.

[3] Shi, F.—Makris, Y.—Nowick, S. M.—Singh, M.: Test Generation for Ultra-High-Speed Asynchronous Pipelines. In: Proceedings of the IEEE International Conference on Test, ITC '05, 2005, pp. 1018–1027.

[4] Cortadella, J.—Kishinevsky, M.—Kondratyev, A.—Lavagno, L.—Yakovlev, A.: Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. IEICE Transactions on Information and Systems, Vol. E80-D, 1997, No. 3, pp. 315–325.

[5] Hauck, S.: Asynchronous Design Methodologies: An Overview. Proceedings of the IEEE, Vol. 83, 1995, No. 1, pp. 69–93.

[6] Brzozowski, J. A.—Iland, Y.—Esik, Z.: Algebras for Hazard Detection. In: Proceedings of the 31st IEEE International Symposium on Multiple-Valued Logic, 2001, pp. 3–12.

[7] Shi, F.—Makris, Y.: SPIN-SIM: Logic and Fault Simulation for Speed-Independent Circuits. In: Proceedings of the 2004 International Test Conference, ITC '04, 2004, pp. 597–606.

[8] Eichelberger, E. B.: Hazard Detection in Combinational and Sequential Switching Circuits. IBM Journal of Research and Development, Vol. 9, 1965, No. 2, pp. 90–99.

[9] Bushnell, M. L.—Agrawal, V. D.: Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits. Kluwer Academic Publishers, New York/Boston/Dordrecht/London/Moscow, 2001.

[10] Novák, O.—Gramatová, E.—Ubar, R. et al.: Handbook of Testing Electronic Systems. Czech Technical University Publishing House, 2005.

[11] Jha, N.—Gupta, S.: Testing of Digital Systems. Cambridge University Press, 2003.

[12] Armstron, D. B.: A Deductive Method for Simulating Faults in Logic Circuits. IEEE Transactions on Computers, Vol. 21, 1972, No. 5, pp. 64–471.

[13] Ulrich, E. G.—Baker, T.: Concurrent Simulation of Nearly Identical Digital Networks. Computer, Vol. 7, 1974, pp. 39–44.

[14] LaFrieda, Ch.—Manohar, R.: Fault Detection and Isolation Techniques for Quasi Delay-Insensitive Circuits. In: Proceedings of the 2004 International Conference on Dependable Systems and Networks, DSN '04, 2004, pp. 41–50.

[15] Shi, F.—Makris, Y.: Enhancing Simulation Accuracy Through Advanced Hazard Detection in Asynchronous Circuits. IEEE Transactions on Computers, Vol. 58, 2009, No. 3, pp. 394–408.

[16] Shi, F.—Makris, Y.: SPIN-TEST: Automatic Test Pattern Generation for Speed-Independent Circuits. In: Proceedings of the 2004 International Conference on Computer Aided Design, 2004, pp. 903–908.

[17] Shi, F.—Makris, Y.: Fault Simulation and Random Test Generation for Speed-Independent Circuits. In: Proceedings of the 2004 Great Lakes Symposium on VLSI, 2004, pp. 127–130.

[18] Chakraborty, T. J.—Agrawal, V. D.—Bushnell, M. L.: Delay Fault Models and Test Generation for Random Logic Sequential Circuits. In: Proceedings of the 29th ACM/IEEE Conference on Design Automation, 1992, pp. 165–172.

[19] Lee, H. K.—Ha, D. S.: On the Generation of Test Patterns for Combinational Circuits. Departmentt of Electrical Eng., Virginia Polytechnic Institute, 1993, Techreport 12_93.

[20] Russel, S. J.—Norvig, P.: Artificial Intelligence – A Modern Approach. Prentice-Hall, Inc., New Jersey, 1995.

[21] Menon, P. R.—Chappell, S. G.: Deductive Fault Simulation with Functional Blocks. IEEE Transactions on Computers, Vol. C-27, 1978, No. 8, pp. 689–695.

[22] Walczak, K.: Deductive Fault Simulation for Sequential Module Circuits. IEEE Transactions on Computers, Vol. 37, 1988, No. 2, pp. 237–239.

[23] Sur-Kolay, S.—Roncken, M.—Stevens, K.—Chaudhuri, P. P.—Roy, R.: Fsimac: A Fault Simulator for Asynchronous Sequential Circuits. In: Proceedings of the 9th Asian Test Symposium, 2000, pp. 114–119.

[24] Myers Research Group. ATACS online demo. University of Utah. Available on: `http://www.async.ece.utah.edu/atacs-bin/demo`, 9/28/2009.

[25] GNU Project. Time. Available on: `http://www.gnu.org/software/time/`, 9/28/2009.

Roland Dobai received the Bachelor's and the Master's degree in computer engineering from the Slovak University of Technology (SUT) Bratislava in 2006 and 2008, respectively. Currently he is a Ph. D. student at the Institute of Informatics of the Slovak Academy of Sciences in Bratislava (Slovakia). His research is targeted at testing of asynchronous sequential circuits. He is a student member of IEEE.

Elena Gramatová graduated from Comenius University in Bratislava, Slovakia (mathematics) and received Ph. D. degree in the technical cybernetics programme from the Slovak Academy of Sciences (SAS) in 1971 and 1984, respectively. She has worked at the Institute of Informatics SAS in Bratislava since 1971 for the Design and Diagnostics of Digital Systems Department. She is a member of IEEE Computer Society (from 2009 is a member of the IEEE Golden Core) and the Slovak contact person of the Test Technology Technical Council. In June 2009 she has started to work as Associate Professor at the Faculty of Informatics and Information Technologies of the Slovak University of Technology in Bratislava. Her research and courses are targeted at testing and reliability of digital systems.