Simulated Injection of Radiation-Induced Logic Faults in FPGAs

Cinzia Bernardeschi, Luca Cassano, Andrea Domenici
Department of Information Engineering
University of Pisa, Italy
Pisa, Italy
first_name.last_name@ing.unipi.it

Giancarlo Gennaro, Mario Pasquariello
Intecs S.p.A.
Pisa, Italy
first_name.last_name@intecs.it

***Abstract*—SRAM-FPGA systems are simulated with a model based on the Stochastic Activity Networks (SAN) formalism. Faults are injected into the model and their propagation is traced to the output pins using a four-valued logic that enables faulty logical signals to be tagged and recognized without recurring to a comparison with the expected output values. Input vectors are generated probabilistically based on assumed signal probabilities. By this procedure it is possible to obtain a statistical assessment of the observability of different faults for the generated inputs. The analysis of a 2-out-of-2 voter is shown as a case study.**

***Keywords*-SRAM-FPGA; Simulation; Single Event Upset; Single Event Transient; Stochastic Activity Networks**

I. INTRODUCTION

In the last decade SRAM-FPGAs played a very important role in the market of silicon devices, thanks to the low cost and relatively good performance. In the last years FPGAs have increasingly been employed also in safety-related applications such as railway signaling [1], radar systems for automotive applications [2] and wireless sensor networks for aerospace [3].

The industrial use of electronic devices in safety-critical systems imposes a rigorous system design and the identification of hazardous failure modes. This is particularly true for programmable electronic devices, such as FPGAs, since the failure modes observable at the boundary of the system strongly depend on the application implemented in the device.

Radiations in the atmosphere are responsible for introducing *Single Event Upsets* (SEU) and *Single Event Transients* (SET) in digital devices [4], [5]. SEUs have particularly adverse effects on FPGAs using SRAM technology, as they may alter a bit in the configuration memory, causing a permanent fault (correctable only with a reconfiguration of the device) [6]. SETs may temporarily alter the behaviour of user resources, such as flip-flops and multiplexers.

In this work, we present a simulation based fault injector for SRAM-FPGA systems that can be used for the analysis of radiation-induced logic faults. The FPGA is considered at the netlist level and SEUs and SETs affecting the logic resources of FPGAs are considered. The simulator is based on a model of SRAM-FPGA systems described with the Stochastic Activity Networks (SAN) formalism [7] and developed with the Möbius tool [8]. Faults are injected into the model and their propagation is traced to the output pins using a four-valued logic (along the lines of the D-calculus [9]) that enables faulty logical signals to be tagged and recognized. Input vectors are generated probabilistically based on assumed signal probabilities. For every generated test pattern (i.e., a sequence of input vectors), each possible fault in the adopted model is injected. By this procedure it is possible to obtain a statistical assessment of the observability of different faults for the given test patterns.

The remainder of this paper is organized as follows: Section II, briefly discusses the state of the art in the FPGA fault injection field; in Section III, the considered fault model is presented; Section IV, shows the SAN formalism and the Möbius tool; in Section V, the model of FPGA-based systems and the fault injector are presented; in Section VI, the simulator engine, the available measures and an example of application are shown; Section VII, concludes the paper.

II. STATE OF THE ART

Fault injection is a widely used approach to evaluate the propagation of faults in digital devices. Fault injection techniques for SRAM-FPGA based systems can be divided into prototype-based [10], [11] and simulation-based [12], [13]. Prototype-based techniques have high performance and accuracy, but, since they are performed at the end of the design process, they make corrections expensive. Additionally they often depend on the particular vendor and model of the FPGA chip. Simulation-based techniques alleviate these problems offering the designer greater observability and controllability, but their accuracy may be limited by the assumptions on the system and fault model.

To the best of our knowledge, simulated fault injection for FPGAs at the netlist level has been proposed only in [12] and in [13], but, unlike our method, these tools are not entirely based on simulation, since they rely on an underlying prototype-based analysis. Further, with respect to both [12] and [13], the four-value logic allows us to recognize faulty signals without recurring either to a golden run or a golden copy of the system. Our choice of considering the system

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-168-7

at the netlist level is due to the fact that at the register-transfer level (i.e., VHDL or Verilog description) faults in the hardware structure of the system can not be analysed. Moreover, the SAN model is quite general and allows different kind of analyses to be performed, such as failure probability computation [14].

III. Fault model

An FPGA is a prefabricated array of programmable blocks, interconnected by a programmable routing architecture and surrounded by programmable I/O blocks [15].

Programming an SRAM-FPGA device consists in downloading a programming code, called a *bitstream*, into its configuration memory. The bitstream determines the functionalities of logic blocks, the internal connections among logic blocks and the external connections among logic blocks and I/O pads. Interconnections are realized internally by routing switches and externally by *I/O buffers*. The most common programmable logic blocks are *lookup tables* (LUT), small memories whose contents are defined by configuration bits.

In this work the FPGA system is modelled at the netlist-level representation produced in the synthesis phase before the place and route. At this level, the elements visible in the model are I/O buffers, LUTs, flip-flops, and multiplexers. We consider both SEUs in the configuration memory of LUTs and I/O buffers and SETs in multiplexers and flip-flops

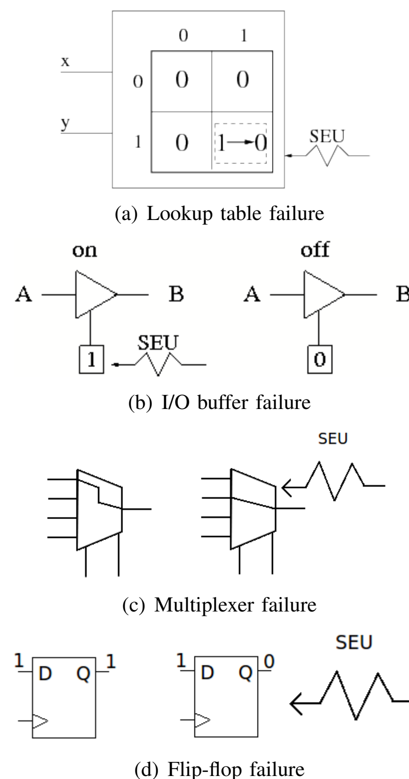
A SEU in the configuration memory of a LUT causes the alteration of the functionality performed by the LUT. Figure 1(a) shows a SEU causing a bit flip in the configuration bit associated to input `(1 1)`. In this case the logic function implemented by the LUT changes from an AND to a constant 0. I/O buffers are connecting resources placed at the input and output of the chip. Each buffer is opened/closed by a configuration bit. A SEU in the configuration bit of a buffer causes an undesired connection/disconnection between two wires, as shown in Figure 1(b).

A SET in a multiplexer causes the temporary selection of a wrong signal, as shown in Figure 1(c). Finally a SET in a memory element, such as a flip-flop (see Figure 1(d)), causes the storage of a wrong value, until a new value is written in the flip-flop.

IV. The SAN Formalism

SANs [7] are an extension of Petri Nets (PN). SANs are directed graphs with four disjoint sets of nodes: *places*, *input gates*, *output gates*, and *activities*. The topology of a SAN is defined by its input and output gates and by two functions that map input gates to activities and pairs (*activity*, *case*) (see below) to output gates, respectively. Each input (output) gate has a set of *input* (*output*) places.

The activities replace and extend the *transitions* of the PN formalism. Any activity may have mutually exclusive outcomes, called *cases*, chosen probabilistically according to the *case distribution* of the activity.



(a) Lookup table failure

(b) I/O buffer failure

(c) Multiplexer failure

(d) Flip-flop failure

Figure 1. Failure modes of various resources of the FPGA chip.

As in PNs, the state of a SAN is defined by its *marking*. The marking of each place is a non-negative integer (called the *number of tokens* of the place).

SANs enable the user to specify any desired enabling condition and firing rule for each activity. This is accomplished by associating an *enabling predicate* and an *input function* to each input gate, and an *output function* to each output gate. The enabling predicate is a Boolean function of the marking of the gate's input places. The input and output functions compute the next marking of the input and output places, respectively, given their current marking.

Graphically, places are drawn as circles, input (output) gates as left-pointing (right-pointing) triangles, and activities as vertical bars. Cases are drawn as small circles on the right side of activities. Gates with default (standard PN) enabling predicates and firing rules are not shown.

A. The Möbius Tool

Möbius [8] is a popular software tool that provides a comprehensive framework for model-based evaluation of system dependability and performance.

SAN models can be composed by means of *Join* and *Rep* operators. Join is used to compose two or more SANs. Rep is a special case of Join, and is used to construct a model consisting of a number of replicas of a SAN. Models composed with Join and Rep interact via *place sharing*. Graphically, a composed model is represented as a tree

Copyright (c) IARIA, 2011.
ISBN: 978-1-61208-168-7

whose nodes are either *atomic* models (i.e., simple SANs), or Join and Rep operators.

Properties of interest are specified with *reward functions*. A reward function specifies how to measure a property on the basis of the SAN marking. Measurements can be made at specific time instants, over periods of time, or when the system reaches a steady state. A desired confidence level is associated to each reward function. At the end of a simulation the Möbius tool is able to evaluate for each reward function whether the desired confidence level has been attained or not thus ensuring a high accuracy of the measurements.

V. MODELLING FPGAS WITH SANS

The FPGA model is split into a number of modules that interact through shared places [16]. Modules *System Manager*, *Input Vector*, *Combinatorial Logic*, and *Sequential Logic* describe the FPGA operation and module *Fault Injector* deals with faults.

The System Manager module orchestrates the activity of the other modules of the system according to the following steps: (i) a fault is injected; (ii) an *input vector*, i.e., an n-tuple of the input signal values, is applied to the input lines; (iii) the combinatorial part of the system is executed; (iv) a clock tick arrives and the sequential part of the system is executed. Steps (ii) through (iv) are repeated until all input vectors have been applied.

The Input Vector module applies an input vector to the input lines of the FPGA.

The Combinatorial Logic module models the combinatorial part of the system. The modelled components are lookup tables, multiplexers, and I/O buffers.

The Sequential Logic module models the flip-flops in the FPGA. Various types of flip-flops can be modelled.

The Fault Injector module is in charge of injecting faults into the netlist. For the purpose of this work, the fault injector injects a single permanent fault into the configuration memory of LUTs and I/O buffers or a single transient fault in the user resources (flip-flops or multiplexers). The fault is injected at the beginning of the simulation. Faults are exhaustively injected in the system one at a time.

Combinatorial and sequential elements are modelled by a SAN model, called `Generic_Component` (see Figure 2(a)). Places `spA` and `spB` are used to control the execution of a component. The output gate `OG0` implements the functionality of the component. When the `execute` activity of a component completes, the function specified in gate `OG0` is executed, and a token is added to `spB`.

Three shared places (`input_lines`, `output_lines`, and `internal_lines`) encode the value of the signals on the input, output, and internal connections of the FPGA. The shared place `faults` keeps track of the faults injected in the system. Components behave correctly or faulty according to the content of place `faults`.

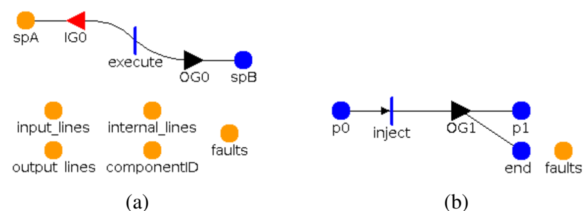


Figure 2. Generic_Component (a) and Fault_Injector (b) module.

The SAN model of the Fault Injector is shown in Figure 2(b). Places `p0` and `p1` are used to control the execution of the fault injector. Place `faults` is shared with the combinatorial logic module. Place `faults` is an array of *C* Boolean values, where C is the number of configuration bits associated to LUTs and I/O buffers, plus the number of flip-flops and multiplexers. In particular `faults[i]` equals 1 if the i-th configuration bit is faulty or the associated user resource is faulty. The output gate `OG1` implements the fault injection function resetting the element of `faults` associated to the previously injected fault and setting the element affected by the new fault. When the `inject` activity completes the function specified in gate `OG1` is executed, and a token is added to `p1`. When every possible faults have been injected and the associated simulation runs have been performed, a token is placed into `end`.

The logical connections are specified in a *connectivity matrix*, a data structure accessed by the input and output functions of the model. This way, the logical connections are not hardwired in the SAN models, and can be set up starting from netlist EDIF file generated by CAD tools, such as the Xilinx ISE tool, on the basis of the specification of the FPGA-based system.

VI. THE SIMULATOR

The simulator executes the previously discussed model of FPGA-based systems on a four-valued logic that enables faulty logical signals to be tagged as such and followed along their propagation path. In this logic we distinguish correct values from faulty ones (as in D-Calculus [9]).

Correct and faulty Boolean values are named zero correct (0_c), one correct (1_c), zero faulty (0_f) and one faulty (1_f). More precisely, let $B = \{0, 1\}$ and $D = \{0_c, 0_f, 1_c, 1_f\}$, where B is the set of standard Boolean values and D is the domain of the four-valued logic. Then we establish a correspondence between B and D by the following mappings:
$\phi : D \rightarrow B$, such that $\phi(0_c) = \phi(0_f) = 0$ and $\phi(1_c) = \phi(1_f) = 1$, is a *projection* function that translates values of D to values in B ignoring the faulty/correct annotation.
$\chi : D \rightarrow B$, such that $\chi(0_c) = 0$, $\chi(1_c) = 1$, $\chi(0_f) = 1$ and $\chi(1_f) = 0$, is a *corrective* projection that replaces a faulty value with its complemented Boolean value (i.e. it extracts a correct value from a faulty one).

Then we define *tracking* functions for components. These

x	y	$\wedge^*$
0_c	-	0_c
-	0_c	0_c
1_c	1_c	1_c
1_c	0_f	0_f
1_c	1_f	1_f
0_f	1_c	0_f
1_f	1_c	1_f
0_f	0_f	0_f
0_f	1_f	0_c
1_f	0_f	0_c
1_f	1_f	1_f

x	y	$\vee^*$
1_c	-	1_c
-	1_c	1_c
0_c	0_c	0_c
0_c	0_f	0_f
0_c	1_f	1_f
0_f	0_c	0_f
1_f	0_c	1_f
0_f	0_f	0_f
0_f	1_f	1_c
1_f	0_f	1_c
1_f	1_f	1_f

x	$\neg^*$
0_c	1_c
1_c	0_c
0_f	1_f
1_f	0_f

function q^*		
D	Q_{prev}	Q
-	0_c	0_c
-	1_c	1_c
-	0_f	0_f
-	1_f	1_f

function q_c^*		
D	Q_{prev}	Q
0_c	-	0_c
1_c	-	1_c
0_f	-	0_f
1_f	-	1_f

Table I
FOUR-VALUED TRUTH TABLES FOR AND, OR, NOT, AND D EDGE-TRIGGERED FLIP-FLOP.

functions trace the propagation of values through components. Each non-faulty component implements a Boolean function $f : B^n \rightarrow B$. For such function, its tracking function $f^* : D^n \rightarrow D$ extends the semantics of f to the four-valued domain D. For a given n-tuple of inputs $(d_1, \cdots, d_n)$ in D^n, this function evaluates f both with the projection of $(d_1, \cdots, d_n)$ to B^n (i.e., $(\phi(d_1), \cdots, \phi(d_n))$) and with the corrective projection of $(d_1, \cdots, d_n)$ to B^n (i.e., $(\chi(d_1), \cdots, \chi(d_n))$). This amounts to applying f to the actual inputs and to the input that would have been applied in absence of faults. Function f^* compares the two results. If they are equal, the result is $f(\phi(d_1), \cdots, \phi(d_n))$ tagged as a correct value, otherwise the result is $f(\phi(d_1), \cdots, \phi(d_n))$ tagged as faulty.

In particular, we define the four-valued logical operators $\wedge^*$, $\vee^*$ and $\neg^*$ as the tracking functions of the corresponding Boolean operators. The semantics of these operators is given by truth tables (see Table I). We may notice that for the $\wedge^*$ operator, a 0_c on one input masks any faulty value on the other input; similarly for the $\vee^*$ operator a 1_c masks any faulty value on the other input.

We defined the tracking function for the components of the netlist: I/O buffers, LUTs, multiplexers, and flip-flops. Flip-flops are modelled with two functions: the first one models the behaviour of the flip-flop in the presence of a clock rising edge (called q_c^*), the other (q^*) models the behavior of the flip-flop during the inactive period. For example, the functions for a standard D-Edge Triggered flip-flop are shown in Table I. We may notice that the output (correct or faulty) is unchanged in absence of a rising edge, while it follows the input when a rising edge occurs.

We now show how to model the generation of faulty values by faulty components and the propagation of values through faulty components. Given a Boolean function $f : B^n \rightarrow B$ implemented by a logic component, for each possible fault i of the component we define a *faulty* function $\hat{f}_i : B^n \rightarrow B$ that describes the behaviour of the component in presence of that fault. This behaviour may be given in the form of truth table or as an expression. For simplicity, in the following we will drop the subscript.

Then, the tracking function $\hat{f}^*$ of a faulty function $\hat{f}$ compares the output of the faulty component with possibly faulty inputs to the output of the correct component with correct inputs. If the two are equal, the result is taken as correct. Otherwise it is tagged as faulty.

For example, given a two-input LUT implementing the AND function, let us assume that a fault occurs in the configuration bit associated with the input $x = 1$ and $y = 1$ (Figure 1(a)). When the input is $(1, 1)$, the output of the LUT is 0 instead of 1 (the output of the faulty LUT is always 0).

Function f_{LUT} describes the behaviour in absence of faults: $f_{LUT}(d_1, d_2) = d_1 \wedge d_2$, whereas function $\hat{f}_{LUT}$ represents the behaviour of the faulty LUT: $\hat{f}_{LUT} = 0$ if $d_1 = 1\ and\ d_2 = 1$, $\hat{f}_{LUT} = f(d_1, d_2)$ otherwise.

Table II shows three cases for the tracking function of the faulty LUT: in the first case, two correct inputs activate the fault and generate a faulty output; in the second case the correct input 1_c and the faulty input 0_f do not activate the fault. However, the resulting output differs from the output that should have been produced with 1 and 1, and the faulty value is propagated. In the third case the correct input 1_c and the faulty input 1_f activates the fault but the resulting output equals the output that should have been produced with 1 and 0. We notice that in the course of simulation, the tracking functions can be calculated off-line and synthesized as truth tables before starting the simulation.

d_1	d_2	$\hat{f}(\phi(d_1), \phi(d_2))$	$f(\chi(d_1), \chi(d_2))$	$\hat{f}^*$
1_c	1_c	0	1	0_f
1_c	0_f	0	1	0_f
1_c	1_f	0	0	0_c

Table II
AN EXCERPT OF THE FAULTY LUT TRUTH TABLE

Using this four-valued logic we are able to trace the propagation of faults and to determine whether they reach the output, and, if not, to find which components mask or propagate the fault. This four-valued logic allows the observability of faults to be measured (a comparison of the actual output values with the expected ones is not necessary).

A. Simulation and Measurements

The configurable parameters of our simulations are the number of simulated clock cycles N and the signal probability of input signals SP_i, i.e. the probability of the signal to be 1 at a given time [17].

In order to measure the fault observability of the system under analysis we perform multiple simulation runs of the system. Each simulation run is structured in the following steps, graphically represented by Figure 3:

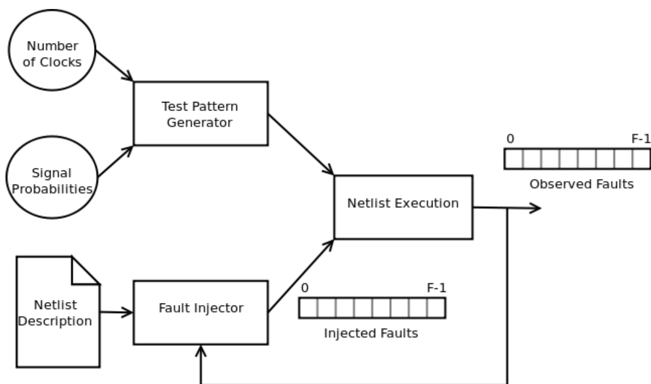


Figure 3. Functional blocks of the simulator.

1) A test pattern is stochastically generated according to given input signal probabilities and a number of clocks.
2) A fault is injected in the system.
3) The netlist is executed until the maximum number of clock cycles is reached. The following reward function detects system failures:

```
if(System_Manager->clock->Mark()==1){
   for(int i=0; i<N_out; ++i)
      if(System_Manager->output_lines->
         Index(i)->Mark() == 1f ||
         System_Manager->output_lines->
         Index(i)->Mark() == 0f)
             return 1;
    return 0; }
```

4) If more faults have to be injected, the current fault is removed and the simulation re-starts from step 2, otherwise simulation terminates.

Data that can be obtained with our analysis are the list of observed faults for each generated test pattern, and the total number of observed faults using the generated test patterns. From these data we can compute a quality factor, called *total observability*, of the set of test patterns, defined as the ratio of observed faults to the total number of injected faults.

The above shown reward function allows the analysis of the observability of faults at the output of the system. Other analyses can be performed: the behavior of any internal signal can be observed and, if a certain fault has been activated and it has not been observed at the output, we can find where the fault has been masked. Moreover, we can model different fault hypotheses, such as multiple faults, or faults confined to a certain area of the device, simply modifying the initialization of the fault injector module.

B. An Example

In order to analyse the applicability of our method we considered as a simple case study an 8-bit 2-out-of-2 voter.

The behaviour of the system is the following:

- After a $0 \rightarrow 1$ transition of Data_Valid, the circuit starts reading serially 8 bits from Stream_A and Stream_B.

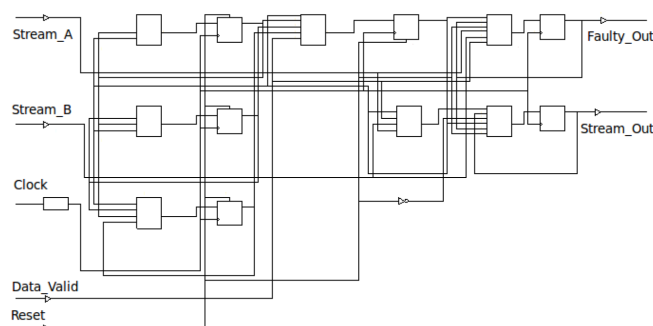


Figure 4. The netlist of the 8 bit 2-out-of-2 voter.

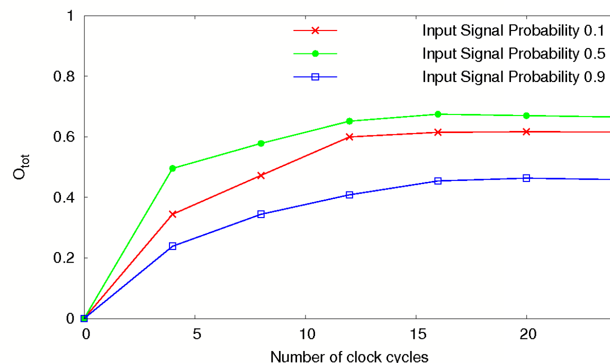


Figure 5. Total observability vs. test pattern length.

- If Stream_A and Stream_B are equal, Stream_Out follows Stream_A and Faulty_Out is 0.
- If Stream_A and Stream_B are different for at least one bit, Stream_Out is set to 0 and Faulty_Out to 1 for the rest of the byte.

We synthesised the system for the Xilinx Virtex 6 device into a netlist with the Xilinx ISE tool. The resulting netlist, (Figure 4), has 4 input signals, 2 output signals, 6 I/O buffers, 8 LUTs, and 6 flip-flops. We then used a parser from EDIF to our specification language to instantiate the model.

In every simulation we set the signal probability of the four input signals of the system to the same value.

In a first scenario we calculated the total observability of the system for $SP = 0.1, SP = 0.5$ and $SP = 0.9$, varying the number of simulated clock cycles. The resulting total observability is shown in Figure 5.

In a second scenario we calculated the total observability of the system for $N = 4, N = 8$ and $N = 12$ clock cycles, varying the signal probabilities of the input signals. The resulting total observability is shown in Figure 6.

Each simulation run took from 0.3 to 0.5 seconds to be carried out. In order to reach a confidence level of 0.95 with a confidence interval of 0.1, we needed from 2000 to 3000 simulation runs. The complete analysis required a few minutes to be performed.

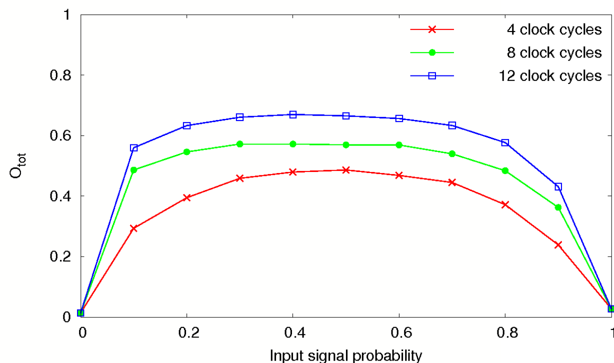


Figure 6. Total observability vs. input signal probability.

VII. Conclusions and Future Work

A simulation based fault injection tool for FPGA systems is shown. The FPGA system is modelled at netlist level. The considered fault model is fine-grained as the effect of SEUs affecting any configuration bit of a LUT and an I/O buffer can be simulated, as well as the effects of SETs in flip-flops and multiplexers. In this work the fault injector has been used for fault observability analysis. These measures can be used for giving details on the places in the logic design where injected faults have been/have not been observed. This information, given as feedback to designers, allows them to increase the system observability by reworking the logic around these places, for example by adding test points for the diagnosis of faults. As future work we intend to analyse the observability of other types of faults, such as faults in the routing architecture. Moreover, we intend to implement the generation of selective test patterns for fault diagnosis.

References

[1] J. Borecky, P. Kubalik, and H. Kubatova, "Reliable Railway Station System Based on Regular Structure Implemented in FPGA," in *Proceedings of the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD '09)*, 2009, pp. 348 –354.

[2] V. Winkler, J. Detlefsen, U. Siart, J. Buchler, and M. Wagner, "FPGA-based Signal Processing of an Automotive Radar Sensor," in *Proceedings of the First European Radar Conference (EURAD)*, 2004, pp. 245 –248.

[3] J. Henaut, D. Dragomirescu, and R. Plana, "FPGA Based High Date Rate Radio Interfaces for Aerospace Wireless Sensor Systems," in *Proceedings of the Fourth International Conference on Systems (ICONS '09)*, 2009, pp. 173 –178.

[4] R. Baumann, "Radiation-induced Soft Errors in Advanced Semiconductor Technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305 – 316, September 2005.

[5] G. Wirth, F. Kastensmidt, and I. Ribeiro, "Single Event Transients in Logic Circuits Load and Propagation Induced Pulse Broadening," *IEEE Transactions on Nuclear Science*, vol. 55, no. 6, pp. 2928 –2935, 2008.

[6] P. Graham, M. Caffrey, J. Zimmerman, D. E. Johnson, P. Sundararajan, and C. Patterson, "Consequences and Categories of SRAM FPGA Configuration SEUs," in *Proceedings of the 6th Military and Aerospace Applications of Programmable Logic Devices (MAPLD'03)*, September 2003, p. n.a.

[7] W. Sanders and J. Meyer, "Stochastic activity networks: formal definitions and concepts," in *Lectures on Formal Methods and Performance Analysis*, ser. Lecture Notes in Computer Science, E. Brinksma, H. Hermanns, and J. Katoen, Eds. Springer Berlin / Heidelberg, 2001, vol. 2090, pp. 315–343.

[8] G. Clark, T. Courtney, D. Daly, D. D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The Möbius modeling tool," in *9th Int. Workshop on Petri Nets and Performance Models*. Aachen, Germany: IEEE Computer Society Press, September 2001, pp. 241–250.

[9] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, vol. 10, no. 4, pp. 278 –291, July 1966.

[10] L. Sterpone and M. Violante, "A New Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 54, no. 4, pp. 965 –970, 2007.

[11] E. Johnson, M. Wirthlin, and M. Caffrey, "Single-Event Upset Simulation on an FPGA," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, June 2002, pp. 68–73.

[12] M. Violante, L. Sterpone, M. Ceschia, D. Bortolato, P. Bernardi, M. Reorda, and A. Paccagnella, "Simulation-Based Analysis of SEU Effects in SRAM-Based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 51, no. 6, pp. 3354 – 3359, December 2004.

[13] G. H. Wang Zhongming, Yao Zhibin and L. Min, "A Software Solution to Estimate the SEU-induced Soft Error Rate for Systems Implemented on SRAM-based FPGAs," *Journal of Semiconductors (Chinese Institute of Electronics)*, vol. 32, no. 5, pp. 1–7, May 2011.

[14] C. Bernardeschi, L. Cassano, and A. Domenici, "Failure Probability of SRAM-FPGA Systems with Stochastic Activity Networks," in *Proceedings of the 14th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, April, pp. 293 – 296.

[15] I. Kuon, R. Tessier, and J. Rose, "FPGA Architecture: Survey and Challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, pp. 135–253, February 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1454695.1454696

[16] C. Bernardeschi, L. Cassano, A. Domenici, and P. Masci, "A Tool for Signal Probability Analysis of FPGA-Based Systems," in *Proceedings of the 2nd International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking*, 2011, in press.

[17] V. Saxena, F. Najm, and I. Hajj, "Estimation of State Line Statistics in Sequential Circuits," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 3, pp. 455–473, 2002.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-168-7