A Tool for Signal Probability Analysis of FPGA-Based Systems

Cinzia Bernardeschi[1], Luca Cassano[1], Andrea Domenici[1] and Paolo Masci[2]
[1] Department of Information Engineering, University of Pisa, Italy
[2] Department of Electronic Engineering and Computer Science, Queen Mary University of London, United Kingdom
Email: {cinzia.bernardeschi, luca.cassano, andrea.domenici}@ing.unipi.it, paolo.masci@eecs.qmul.ac.uk

***Abstract*—We describe a model of Field Programmable Gate Array based systems realised with the Stochastic Activity Networks formalism. The model can be used (i) to debug the circuit design synthesised from the high level description of the system, and (ii) to calculate the signal probabilities and transition densities of the circuit design, which are parameters that can be used for reliability analysis, power consumption estimation and pseudo random testing. We validate the developed model by reproducing the results presented in other studies for some representative combinatorial circuits, and we explore the applicability of the proposed model in the analysis of real-world devices by analysing the actual implementation of a circuit for the generation of Cyclic Redundancy Check codes.**

***Keywords*-FPGA, Signal Probability, Simulation, Transition Density.**

I. Introduction and Related Works

Field Programmable Gate Array (FPGA) devices are widely used components in many different application fields, including safety-critical systems. Especially in embedded and mobile applications, the assessment of such quality factors as power consumption and reliability is of fundamental importance. It has been shown that these factors may be estimated in terms of *signal probability* [1], [2], [3], [4], that can be defined as the fraction of clock cycles in which a given signal is high [5]. Another useful parameter is *transition density*, i.e., the fraction of clock cycles in which a signal makes a transition [6]. Other applications of signal probabilities are *soft error rate* estimation [7] and *random testing* [8]. Soft error rate is the error rate due to Single Event Upsets, i.e., errors caused by radiations, that are a major threat to system reliability. In random testing, test patterns are generated at random to cover as many as possible fault modes of the system. The statistical distribution of the test patterns may be weighted according to the input signal probabilities to optimise the coverage.

The computation of signal probabilities may rely on either an analytical or a simulative approach. With analytical models, exact values of signal probabilities can be computed, but the computation is NP-hard in the general case [9], so it is usually necessary to resort to heuristic approximations. With a simulative approach, a model of the system is fed with inputs whose values reflect the expected statistical properties, and the simulated output signals are recorded and analysed to evaluate the resulting properties.

In this paper, we present a model of FPGA circuit execution that can be used to calculate the signal probabilities and transition densities of a given FPGA design, starting from the signal probabilities of the inputs. The model is based on the formalism of *Stochastic Activity Networks* (SAN) [10] and it has been developed with the Möbius tool [11].

In FPGA systems, a high-level design is implemented with the configurable logic blocks made available by a given FPGA chip. In order to attain a realistic model and satisfactory accuracy of the analysis, the proposed model represents the FPGA system at this implementation level.

The model is implemented by a simulator that takes as input a description of the system to be simulated and a few configuration parameters, including the signal probabilities of the inputs, the number of simulated clock cycles etc. The simulator generates input vectors according to the specified signal probabilities of the inputs and the results are collected and analysed using the features of the underlying Möbius environment.

In the rest of this paper, the FPGA technology (Section II) and the SAN formalism (Section III) are introduced, then the formal model of FPGA circuit execution is presented (Section IV) and a case study is shown as a proof of concept (Section V). Conclusions and future work are in Section VI.

II. The FPGA Technology

An FPGA is an array of programmable logic blocks, interconnected through a programmable routing architecture and communicating with the output through programmable I/O pads [12]. The programming of an FPGA device consists in downloading a programming code, called *bitstream*, in its configuration memory, that determines the hardware structure of the system to be implemented in the FPGA, and thus the functionality performed by the system.

The logic blocks may be simple combinatorial/sequential functions, such as lookup tables, multiplexers and flip-flops, or more complex structures such as memories, adders, and micro-controllers. The routing architecture in an FPGA consists of wires and programmable switches that form the desired connections among logic blocks and I/O pads. Finally, the I/O architecture is composed of I/O pads disposed along the perimeter of the device, each one implementing one or more communication standards.

An FPGA system is described at the *Register-Transfer Level (RTL)* in terms of high-level registers and logic functions, independent of their implementation on a particular device.

An RTL specification is usually given in a hardware definition language (HDL), such as VHDL or Verilog. At the *netlist* level, a system is described in terms of its actual implementation, targeted at a particular device and composed of the logic blocks made available by the device, such as lookup tables and flip-flops.

The implementation of an FPGA-based application involves three main phases: (i) the RTL level design is specified with schematics or with a textual description in a HDL; (ii) after an FPGA chip has been selected, a chip-specific tool synthesises the RTL description into a *netlist*, i.e., a textual description of the network implementing the design; and (iii) the bitstream is generated from the netlist.

III. The SAN Formalism

Stochastic Activity Networks [10] are an extension of the Petri Nets (PN). SANs are directed graphs with four disjoint sets of nodes: *places*, *input gates*, *output gates*, and *activities*. The latter replace and extend the *transitions* of the PN formalism. 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.

Each SAN activity may be either *instantaneous* or *timed*. Timed activities represent actions with a duration affecting the performance of the modelled system, e.g., message transmission time. The duration of each timed activity is expressed via a *time distribution* function. Any instantaneous or timed activity may have mutually exclusive outcomes, called *cases*, chosen probabilistically according to the *case distribution* of the activity. Cases can be used to model probabilistic behaviours. An activity *completes* when its (possibly instantaneous) execution terminates.

As in PNs, the state of a SAN is defined by its *marking*, i.e., a function that, at each step of the net's evolution, maps the places to non-negative integers (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. If these predicates and functions are not specified for some activity, the standard PN rules are assumed.

The evolution of a SAN, starting from a given marking μ, may be described as follows: (i) The instantaneous activities enabled in μ complete in some unspecified order; (ii) if no instantaneous activities are enabled in μ, the enabled (timed) activities become *active*; (iii) the completion times of each active (timed) activity are computed stochastically, according to the respective time distributions; the activity with the earliest completion time is selected for completion; (iv) when an activity (timed or not) completes, one of its cases is selected according to the case distribution, and the next marking μ' is computed by evaluating the input and output functions; (v) if an activity that was active in μ is no longer enabled in μ', it is removed from the set of active activities.

Graphically, places are drawn as circles, input (output) gates as left-pointing (right-pointing) triangles, instantaneous activities as narrow vertical bars, and timed activities as thick 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 [11] is a popular software tool that provides a comprehensive framework for model-based evaluation of system dependability and performance. The Möbius tool introduces *shared variables* and *extended places* as extensions to the SAN formalism. Shared variables are global objects that can be used to exchange information among modules. Extended places are places whose marking is a complex data structure instead of a non-negative integer. Enabling predicates and input and output functions of the gates are specified as C++ code.

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*.

Properties of interest are specified with *reward functions*. A reward function specifies how to measure a property on the basis of the SAN marking. There are two kinds of reward functions: *rate reward* and *impulse reward*. Rate rewards can be evaluated at any time instant. Impulse rewards are associated with specific activities and they can be evaluated only when the associated activity completes. Measurements can be conducted at specific time instants, over periods of time, or when the system reaches a steady state.

IV. Description of the Model

The model is split into a number of modules that interact through place sharing. Each module identifies a different logical component of the FPGA: *System Manager* co-ordinates the logical components; *Input Vector* models the signals applied to the input pins; *Combinatorial Logic* models the memoryless circuits; *Sequential Logic* models the storage elements.

The communication among modules reflects the logical connections of the real FPGA components. The logical connections are specified in a *connectivity matrix*, which is a parameter of the model. This way, the logical connections are not hardwired in the SAN models, and can be set up from a text file generated with software tools, such as the Xilinx ISE tool [13], on the basis of the specification of the FPGA.

The overall FPGA system model is shown in Figure 1. The models of the logical components are represented with labelled dark boxes, and their composition is obtained through the *Join* and *Rep* operators. Each model, except *System Manager*, is obtained by composing a number of replicas of customisable template models. Each replica is uniquely identified by an

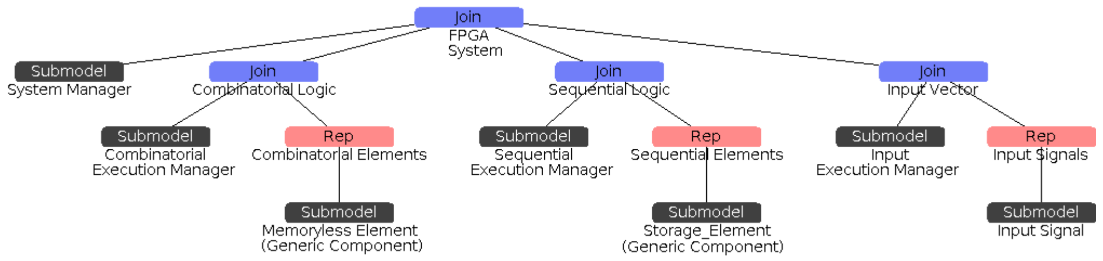


Fig. 1. SAN model of the FPGA.

integer number. The co-ordination between *System Manager* and replicas is accomplished through *Execution Manager*s.

A. System Manager

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

The SAN model of the *System Manager* is shown in Figure 2. The state of the modelled FPGA is given by the marking of three shared places (`input_lines`, `output_lines`, and `internal_lines`), which are vectors that encode the value of the signals on the input, output, and internal lines of the FPGA. Information on the occurrence of transitions on the lines are also maintained in the model with the shared places `input_trans`, `output_trans`, and `internal_trans`. The state includes also two flags: `steady_state_flag`, whose marking reports if the model has reached a steady state; `error_flag`, whose marking reports if the model reaches abnormal execution conditions, e.g., instability of the combinatorial circuit. These flags can be used by analysts and developers to check the consistency of the model specification and to detect potential design problems in the FPGA.

The initial marking of *System Manager* is the following: places `input_lines`, `output_lines`, and `internal_lines` are set according to an initial state of the system; place `signal_length` contains a number of tokens equal to the number of input vectors that will be applied; place `p0` contains one token; all other places hold zero tokens.

Initially, the instantaneous activity `apply_inputs` is enabled because `p0` contains one token. When `apply_inputs` completes, the application of an input vector is triggered by activating module *Input Vector* (Section IV-B). The activation of the *Input Vector* module is obtained by moving the token stored in `p0` into the shared place `sp0_0`. When the application of the input vector completes, module *Input Vector* moves a token into `sp1_1`, and the instantaneous activity `executeCC` of the *System Manager* becomes enabled.

When activity `executeCC` completes, the execution of the combinatorial elements of the model starts by activating module *Combinatorial Logic* (Section IV-C). The activation of this module is obtained by moving the token stored in `sp1_1` into the shared place `sp2_0`. When the execution of the combinatorial elements completes, the *Combinatorial Logic* module moves a token into `sp3_1`, thus enabling the timed activity `executeSC` of the *System Manager*.

When the timed activity `executeSC` completes, a clock tick has arrived, and the execution of the sequential elements starts by activating module *Sequential Logic* (the token stored in `sp3_1` is moved into `sp4_0`). When the execution of the sequential elements completes, the *Sequential Logic* module (Section IV-C) moves a token into `sp5_1`. If the system has reached a steady state, a token is also moved into `steady_state_flag`.

At this point, the instantaneous activity `finalise` is enabled. When `finalise` completes, the marking of the model is updated according to the following three cases: (i) `signal_length` holds more than one token; in this case, the marking of `p0` is incremented by one; this marking triggers the application of a new input vector; (ii) `signal_length` holds zero tokens and the system state is not steady (i.e., `steady_state_flag` contains zero tokens); in this case, a token is moved into `sp1_1`; this marking triggers a new execution of the combinatorial and sequential parts of the model; (iii) `signal_length` has zero tokens and `steady_state_flag` has one token; in this case, the system has reached the final steady state and the execution terminates; a token is moved into `p3` and no activity will be further enabled in the model.

B. Input Vector

The *Input Vector* module applies an input vector to the input lines. The elements of the input vector are generated according to the signal probability of the corresponding signal. The total number of input lines is a model parameter (N_{in}).

The module consists of a manager sub-module, which co-ordinates the concurrent execution of activities in the model, and a number of customised template model replicas, each of which applies an input value to an input line.

1) Execution Manager: This sub-module co-ordinates the parallel execution of the *Input Signal* replicas. The SAN model is shown in Figure 3(a).

In the model, all places initially contain zero tokens, except the shared places that model the FPGA state (`input_lines`, `output_lines`, and `internal_lines`); places `sp0` and `sp1` coincide (through renaming in the Join operator) with

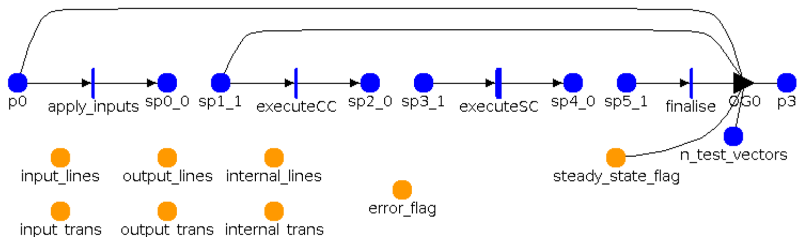
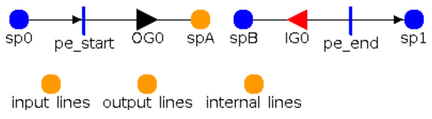
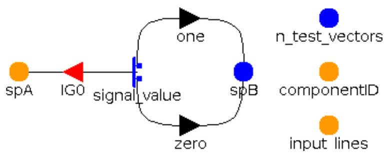


Fig. 2. SAN Model of the System Manager.

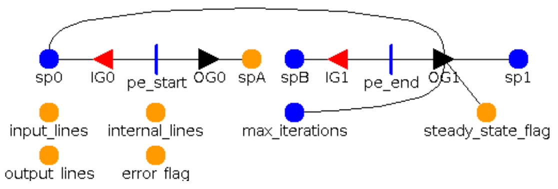


(a) Execution Manager.

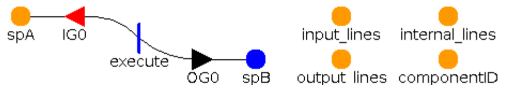


(b) Input Signal.

Fig. 3. Sub-modules of the Input Vector SAN model.



(a) Iterative Execution Manager.



(b) Generic Component.

Fig. 4. Sub-modules of the Combinatorial and Sequential Logic SAN model.

places `sp0_0` and `sp1_1` of *System Manager*, and are thus shared between the two sub-modules. Places `spA` and `spB` are shared with the *Input Signal* sub-modules; `spA` is a vector of N_{in} Booleans; one token in position i encodes a $true$ value for the corresponding element, and triggers the i-th replica of *Input Signal*.

Activity `pe_start` is enabled when *System Manager* moves a token into `sp0`. When the activity completes, the parallel execution of the *Input Signal* replicas starts: gate `OG0` is executed, and N_{in} tokens are moved in the shared vector `spA` (one token for each element of the vector). When the parallel execution of the *Input Signal* replicas concludes, the shared place `spB` contains N_{in} tokens, and the input gate `IG0` enables `pe_end`, which completes immediately. When `pe_end` completes, all tokens stored in `spB` are removed, and one token is moved into `sp1` to signal *System Manager* that the input lines have been updated with the input vector.

2) Input Signal: This sub-module models the application of a signal value at an input line. The SAN model is shown in Figure 3(b). We exploit the semantics of the case probabilities to specify the signal value in terms of the probability of having a logical zero or a logical one.

The *Input Signal* sub-module is replicated N_{in} times, in order to have one sub-module instance for each input line. Replicas have unique identifiers, which are used to associate each replica to an input line.

All places of the model are initially empty, except `componentID`, whose marking specifies the identifier of the sub-module instance. Activity `signal_value` of the sub-module with identifier i is enabled when the *Execution Manager* moves a token in the i-th position of `spA`. When `signal_value` of replica i completes, one of the two output gates is executed to apply a signal value to input line i, and a token is added to element i of `spB`.

C. Combinatorial and Sequential Logic

The *Combinatorial Logic* and the *Sequential Logic* modules define the execution of the memoryless elements and the storage elements, respectively, of the FPGA.

Similarly to *Input Vector*, both models consist of a manager sub-module and a number of customised template model replicas, each of which models the functionalities of an elementary component in the FPGA (either a memoryless element or a storage element).

1) Iterative Execution Manager: This sub-module coordinates the parallel execution of the sub-module replicas. This module is an iterative version of the *Execution Manager* sub-module used in *Input Vector*. This iterative version is used to repeatedly activate the parallel execution of the components until either the modelled elements reach a steady state, or a maximum number of iterations has been performed.

The SAN model of the *Iterative Execution Manager* is shown in Figure 4(a). In the model, all places initially contain zero tokens, except the shared places that model the FPGA state (`input_lines`, `output_lines`, and `internal_lines`), and `max_iterations`, whose marking specifies the maximum number of iterations needed to complete the execution of the modelled elements. In the case of combinatorial elements, the maximum number of iterations depends on the interconnection among elements; in the case of sequential elements, the number of iterations is always one.

We describe the *Iterative Execution Manager* in conjunction with the *Combinatorial Elements* module, as the same description applies also to the *Sequential Elements* module.

In the model of the *Iterative Execution Manager*, shared places `sp0` and `sp1` coincide (through renaming in the Join operator) with the shared places `sp2_0` and `sp3_1` of *System Manager*, and are thus shared between the two submodules. Places `spA` and `spB` are shared with the *Combinatorial* submodules. Moving one token in position i of `spA` encodes a $true$ value for the element in position i, and triggers the execution of the i-th replica of *Input Signal*. The total number of combinatorial elements is a model parameter (N_c).

Activity `pe_start` is enabled when *System Manager* moves a token into `sp0` and `max_iterations` contains at least one token. When the activity completes, the parallel execution of the combinatorial elements starts: gate `OG0` is executed, the marking of `max_iterations` is decremented by one, and N_c tokens are moved in the shared vector `spA` (one token for each element of the vector). When the parallel execution of the combinatorial elements concludes, the shared place `spB` contains N_c tokens, and the input gate `IG0` enables `pe_end`, which completes immediately. When `pe_end` completes, all tokens stored in `spB` are removed, and one token is moved into `sp1` to signal *System Manager* that the execution of the combinatorial elements has completed.

2) Combinatorial and Sequential Elements: These elements are modelled through a customisable SAN model, denominated `Generic_Component` (Figure 4(b)).

The set of combinatorial (sequential) elements is obtained by replicating N_c (N_s) times the *Generic Component* model, where N_c (N_s) is the total number of combinational (sequential) elements. Each replica has a unique identifier, used to specify the functionality of each model instance.

All places of the model are initially empty, except `componentID`, which specifies the replica identifier, and `input_lines`, `output_lines`, and `internal_lines`, representing the current system state. The instantaneous activity `execute` of replica i is enabled when the *Iterative Execution Manager* moves a token in the i-th position of `spA`. When the `execution` activity of replica i completes, the function specified in gate `OG0` is executed, and a token is added to `spB`.

V. Analysis

This section presents the results obtained through the simulation of the proposed SAN models using the Möbius [11] tool. The goal of the presented analysis is two-fold: (i) to validate the developed model for the FPGA, by reproducing the results presented in other studies for some representative combinatorial circuits; (ii) to explore the applicability of the proposed model in the analysis of real-world devices, by analysing the actual implementation of a circuit for the generation of Cyclic Redundancy Check (CRC) codes. CRC is a widely used error-detection scheme used in data communication systems to contrast communication failures due to the unreliable nature of the physical links between devices. When data needs to be reliably transmitted over an unreliable link, the sending device includes a CRC field in the transmitted message; this way, the receiving device can check if the received message is damaged and, in such case, arrange for a message retransmission.

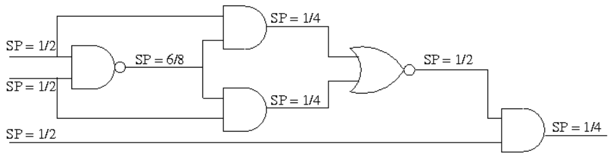


Fig. 5. Example of combinatorial circuit used for validation.

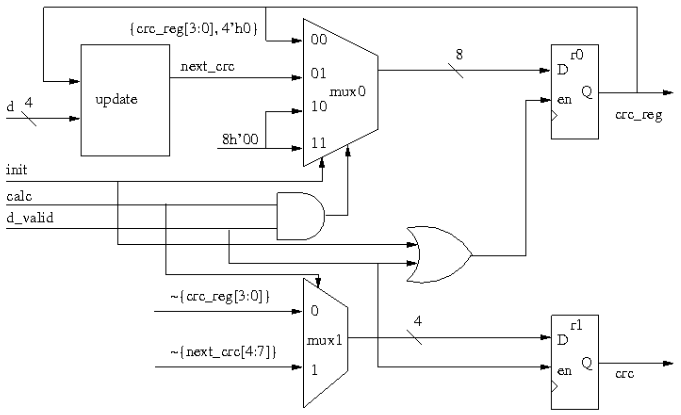


Fig. 6. A circuit to generate CRC code (adapted from [15]).

A. Validation of the Model

To validate our model, we considered the combinatorial circuits presented in various related works and we checked that we were able to reproduce the same results. Let us consider, as a representative case, the combinatorial circuit of Figure 5, which has a reconvergent fanout. The analytical results for signal probability are reported in [14] (we show such results on the upper side of the lines).

The signal probabilities computed with our model correspond with those reported in [14]; specifically, after 10000 simulation runs, we obtained an average relative error of $2 \cdot 10^{-4}$, never exceeding $6 \cdot 10^{-4}$. This cross-validation exercise reinforced our confidence in the correct definition of the model.

B. Analysis of a Circuit for CRC generation

As a case study, we consider the FPGA implementation of a circuit for the generation of IEEE 802.3 CRC codes [15]. A simplified schematic of a 4-bit data bus circuit that generates 8-bit CRC codes is shown in Figure 6. In the figure, `d` is the 4-bit data bus, `init`, `calc`, and `d_valid` are control signals, and `update` is a combinatorial network that computes the next state for the output register `r0`.

The Verilog code for the circuit, which is publicly available from the Xilinx web site, was compiled into a netlist with the Xilinx ISE tool [13]. The resulting netlist has 8 input signals, 12 output signals, 20 matching I/O buffers, 17 LUTs, and 19 flip-flops. We modelled the netlist according to the method described in Section IV and we used the model to compute the signal probabilities and transition densities of the signals on the internal lines of the circuit. The experiments have been set up to reproduce the signal values during a CRC calculation: the control pins `load_init` and `reset` are always low; `calc` is always high; `d_valid` switches between high and low levels at each clock cycle; input pins `d[3:0]` are high with probability P_i, where i is the index of the input pin.

The measurements have been obtained as follows: (i) A simulation run consists in applying a number n_r of consecutive test vectors, where r identifies the run; the test vectors elements are generated with a uniform probability distribution, assuming independence between elements and between vectors; (ii) for each signal i, we defined two reward functions: p_{ri}, which returns, for each run r, the number of clock cycles in which the i-th signal is high; t_{ri}, which returns, in each run r, the number of clock cycles in which the signal makes a transition. The signal probability P_{ri} and transition density T_{ri} of signal i for run r are then computed by dividing p_{ri} and t_{ri}, respectively, by n_r.

We obtain a quantitative assessment of signal probability and transition density for every line of the circuit. The number of test vectors used for the experiment is $n_r = 48$, which corresponds to the length of an IEEE 802.3 address field. The number of simulation runs needed to obtain a confidence level of 95% for this circuit was between $5K$ and $10K$. The time needed to execute $1K$ simulation runs of the model on a 2.67 GHz Intel Core i5 was about 2 minutes.

Some results are shown in Figure 7. The plots shown in the figure report (on the y axis) the value of signal probability and transition density of three internal lines connected to multiplexer `mux0` when varying (on the x axis) the signal probability of the input lines `d[3:0]`. In order to simplify the presentation of the results, in the experiments we imposed that the signal probability varies identically on all input lines.

VI. Conclusions and Future Work

A general model for the execution of FPGA circuits at the netlist level has been defined with the SAN formalism and a simulator has been developed with the Möbius tool. The proposed model is suitable (i) to debug the actual FPGA circuit design synthesised from the high level description of the system and (ii) to compute signal probabilities and transition densities of the design. It is worth noting that, even if the analysis has focused on FPGA systems, the model is applicable to general sequential circuits.

As further work, we intend to extend the model to study system reliability with fault-injection techniques, and we intend to exploit the capabilities of the SAN formalism to model both independent or correlated faults.

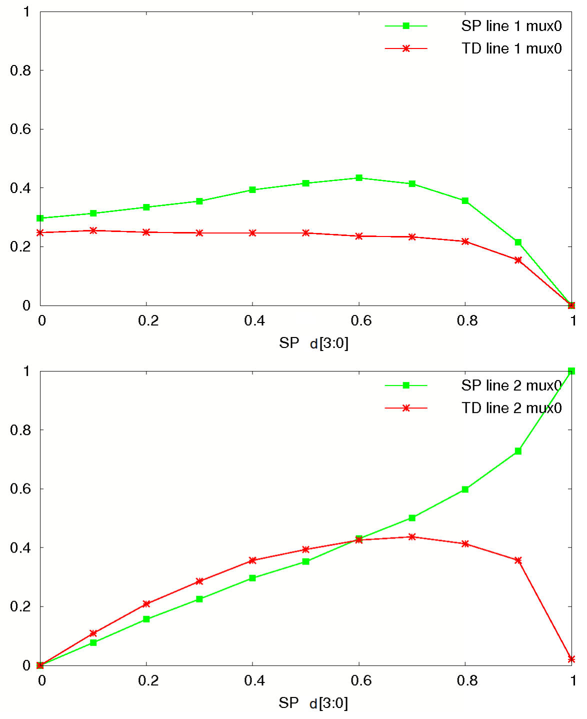


Fig. 7. Example of results for signal probability and transition density of two input lines of mux0 for input data signal probability 0.5.

References

[1] F. N. Najm, "A survey of power estimation techniques in vlsi circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 4, pp. 446–455, december 1994.

[2] D. Franco *et al.*, "Signal probability for reliability evaluation of logic circuits," *Microelectronics Reliability*, vol. 48, no. 8-9, pp. 1586–1591, 2008.

[3] J. T. Flaquer *et al.*, "Fast reliability analysis of combinatorial logic circuits using conditional probabilities," *Microelectronics Reliability*, vol. 50, no. 9-11, pp. 1215–1218, 2010.

[4] H. Chen and J. Han, "Stochastic computational models for accurate reliability evaluation of logic circuits," in *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, ser. GLSVLSI '10. ACM, 2010, pp. 61–66.

[5] K. Parker and E. McCluskey, "Analysis of logic circuits with faults using input signal probabilities," *IEEE Transactions on Computers*, vol. C-24, no. 5, pp. 573–578, may 1975.

[6] V. Saxena *et al.*, "Estimation of state line statistics in sequential circuits," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 3, pp. 455–473, july 2002.

[7] H. Asadi *et al.*, "Soft error susceptibility analysis of SRAM-based FPGAs in high-performance information systems," *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, pp. 2714–2726, dec 2007.

[8] J. Savir, "Distributed generation of weighted random patterns," *Computers, IEEE Transactions on*, vol. 48, no. 12, pp. 1364–1368, Dec. 1999.

[9] B. Krishnamurthy and I. Tollis, "Improved techniques for estimating signal probabilities," *IEEE Transactions on Computers*, vol. 38, no. 7, pp. 1041–1045, Jul. 1989.

[10] W. H. Sanders and J. F. Meyer, "Stochastic activity networks: formal definitions and concepts." New York, NY, USA: Springer-Verlag New York, Inc., 2002, pp. 315–343.

[11] G. Clark *et al.*, "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.

[12] I. Kuon *et al.*, "Fpga architecture: Survey and challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, 2008.

[13] "ISE Design Suite Software Manuals and Help," http://www.xilinx.com/support/documentation/sw_manuals, 2010.

[14] M. Al-Kharji and S. Al-Arian, "A new heuristic algorithm for estimating signal and detection probabilities," in *Proceedings. Seventh Great Lakes Symposium on VLSI, 1997*, mar 1997, pp. 26–31.

[15] C. Borrelli, "IEEE 802.3 Cyclic Redundancy Check," application note: Virtex Series and Virtex-II Family, XAPP209 (v1.0), March 23, 2001, Xilinx, Inc.