Analog Fault Testing Through Abstraction

Enrico Fraccaroli and Franco Fummi
Department of Computer Science, University of Verona, Italy, Email: name.surname@univr.it

***Abstract*—Despite analog SPICE-like simulators have reached their maturity, most of them were not originally conceived for simulating faulty circuits. With the advent of smart systems, fault testing has to deal with models encompassing both analog and digital blocks. Due to their complexity, the industry is still lacking of effective testing approaches for these analog and mixed-signal (AMS) models. The current problem is the computational time required for implementing an analog fault simulation campaign. To this end, the work presented in this paper is an automatic procedure which: *1)* injects faults in an analog circuit, *2)* abstracts both faulty and fault-free models from the circuit to the functional level, *3)* builds an efficient fault simulation framework. The processes of fault injection, faulty model abstraction and framework generation are reported in details, as well as how simulation is carried out. This abstraction process, which preserves the faulty behaviors, allows to reach a speed-up of some orders of magnitude and thus, making feasible an extensive analog faults campaign.**

***Index Terms*—Analog Circuits, Electronic Circuits, Analog Abstraction, Fault Injection, Fault Testing Framework**

I. Introduction

Analog and mixed-signal fault simulation is almost beyond the reach of common computer-aided design (CAD) tools due to the overwhelming complexity of today's integrated circuits (IC). This has induced researchers to rely on high-level models of analog circuits, usually described by using hardware description languages (HDLs) [1], [2]. As a result, the development of fault models at higher levels have achieved great successes, *e.g.*, stuck-at, stuck-open, delay faults belonging to the logic domain. Although their notable performance is compelling, however such models are approximations of real circuit faults and cannot model all of their behaviors [3].

Dependability of electronic devices is becoming harder to achieve and much more crucial with current advancements in the field of nano-scaled semiconductors. Unfortunately, analog fault testing has not reached the same maturity of the digital counterpart due to some major issues. First, most of the analog and mixed-signal simulations are still performed with low-level SPICE-like simulators [4]. They gain in accuracy what they lose in simulation efficiency. Last but not least, analog faults can be caused by many different factors (*e.g.*, impurities, defects, *etc.*) and can cause different faulty behaviors, making fault testing even harder.

The efficient simulation of analog and mixed-signal components has been tackled in [5]. It speeds up the simulation of a subset of the model's values by producing an efficient high-level description (*e.g.*, C++, SystemC). The idea here is to refine and extend that process, so that it can be applied to analog faults simulation. Figure I gives an overview of the proposed testing flow. It automatically injects faults, abstracts the different mutated descriptions and then recombines them to build a complete testing framework. This abstraction process, which preserves the faulty behaviors, allows to reach a speed-up of some orders of magnitude and thus, making feasible an extensive analog fault campaign.

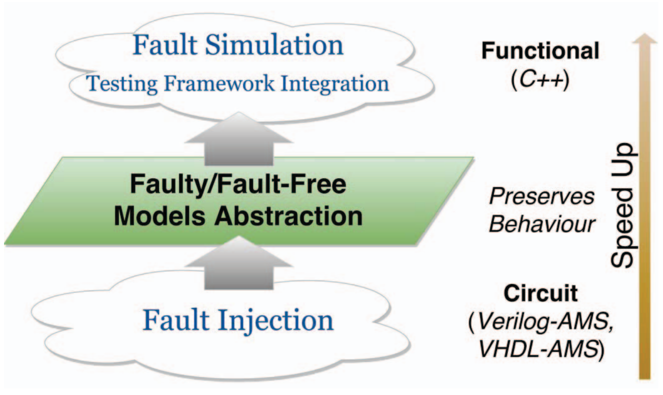

Figure I: Overview of the injection methodology.

II. Background

Fault injection is a technique used to estimate the dependability properties of a device under test (DUT) which comprises the insertion of a fault inside it and the analysis of the consequent response to that particular fault.

An *analog fault* can be seen as a physical alteration of a design while an *analog fault model* is a mathematical description of how the fault changes a design behavior. Both fault models and the design on which it is used must be described at the same level of abstraction. A taxonomy of the different analog hardware description levels with which an analog circuit can be described is depicted in Table 15.1 in [8] by Scheffer *et al.* (2006). It depends on the designer deciding the trade-off between accuracy and efficiency provided by the selected abstraction level.

Fault modeling at *functional* and *behavioral* levels has been explored in [1], [9]. A method for injecting a large number of faults at the *macromodel* level is presented in [10]. At circuit level there are different types of fault models such as open and short circuit, parameter deviation, unwanted contacts, current pulses and so forth. The experimental results presented in this work make use of the double exponential current pulse fault model proposed in [11].

III. Testing Through Abstraction

The structure of the proposed testing framework generation flow is shown in Figure II. The steps of the flow are represent by green boxes while design models as blue boxes with a folded corner. The generation flow accepts as input model an AMS description of a DUT. Without lack of generalization, Verilog-AMS semantics is considered as reference for AMS descriptions [12]. The symbols *ddt* and *idt* are respectively the derivative and integral operators. The Verilog-AMS description shown in Listing I is used as guiding example.

A. Fault injection

The propsoed methodology acquires in the first step the circuit topology (*i.e.*, nodes and branches) of the analog model

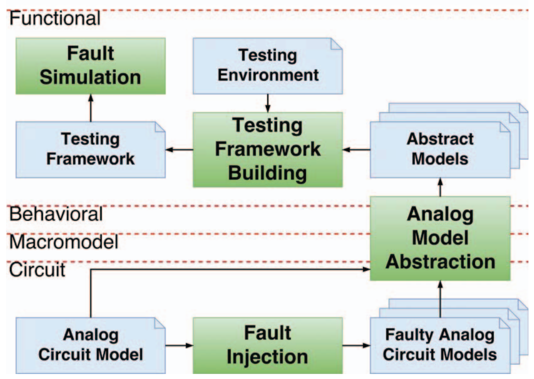

Figure II: Overview of the testing framework generation flow.

and then injects the selected type of fault inside it. In Verilog-AMS the behavior of an analog model is defined as a list of differential equations implemented by using the *branch contribution statement*. A contribution statement is defined by using the *branch contribution operator* (*i.e.*, <+) which is composed by a left-hand side (LHS) and a right-hand side (RHS). The former specifies the physical quantity of a branch to which the value of the latter will be assigned. There are two categories of physical quantities associable to analog nodes: *potential* and *flow*. These can be accessed by means of an *Access Function*, which accepts as arguments two analog nodes and returns the potential difference or the flow between them (*e.g.*, functions `V` and `I` in Listing I). Inside the Verilog-AMS semantics there is a rule that state that: whenever an access function used between two nodes, it implicitly defines a branch between them. For instance, the access function `I(in,n1)` on line 8 of Listing I implicitly defines a branch between the nodes in and $n1$. The topology of the circuit can be retrieved by applying such a rule to all the access functions contained inside a Verilog-AMS model.

The knowledge about the topology allows to inject all types of faults represented by saboteurs and mutants that mutate the structure of the circuit. This is done by adding a new equation to the analog model. A *short circuit* can be modeled by injecting a 1 Ω resistor between the pair of nodes of an existing branch. An *open circuit* can be modeled by injecting a 10 MΩ resistor in series with an existing branch. A *current pulse* can be modeled by injecting a controlled current source with the equation

```
I(node1,node2) <+ pulse;
```

where $pulse$ is a variable used during the simulation to model the desired behavior of the pulse. Also components parameters can be injected by inducing a subtle deviation from designer's

```
1  module OpAmp (in, out);
2      input in; output out;
3      electrical in, out, n1, n2, gnd;
4      ground gnd;
5      parameter real r1    = 400;
6      parameter real c1    = 1e-04;
7      analog begin
8          I(in,n1)    <+ V(in,n1) / r1;
9          I(n1,out)   <+ ddt(V(n1,out)) * c1;
10         I(n1,gnd)   <+ V(n1,gnd) / 1e09;
11         V(n2,gnd)   <+ V(n1,gnd);
12         I(n2,out)   <+ V(n2,out) / 1e03;
13         I(out,gnd)  <+ V(out,gnd) / 1e09;
14     end
15 endmodule
```

Listing I: Operational amplifier Verilog-AMS code.

specifications. The fault model chosen for this work is the current pulse, placed between each internal node and the ground node. If applied to the circuit of Listing I, the list of injectable nodes are: n_1, n_2 and out. The fault injection step generates a faulty analog circuit model for each fault.

B. Analog model abstraction

The second step of the generation flow implements the process of abstraction to both fault-free and mutated descriptions. By referring to the abstraction levels reported in [8], the abstraction used in this paper is a process that transforms an analog model at the *circuit level* to a model at the *functional level*. The guiding idea of the abstraction is to automatically transform an analog model to a signal flow description by finding the input-output relations of a subset of the model values. These values, will be referred from now on as *values of interest*. The automatic definition of such values is a key aspect for applying the abstraction to the faulty models.

1) Equation system enrichment: This step has the purpose of gathering all the branches equations implied by the two principles of conservation of *energy* and *charge*. Applied to the electrical domain these equations are described by the first and second circuit laws of Kirchhoff applied respectively to nodes and loops. The equations relating the same node or the same loop are in a relation of linear dependency.

2) Cone of influence exploration: This phase derives the signal flow description of each value of interest by using the system of equations enriched at the previous step. Such description is built by selecting one of the possible *subsets of equations* describing the behavior of the value of interest *w.r.t.* the inputs of the model. This is done by means of a graph representation of the system of equations. A node is associated to each equation and labeled with LHS variable of the equation. Then, an edge connects a node A to a node B whenever the LHS variable of the equation associated with the second node (B) appears on the RHS of the equation associated with the first node (A). The graph is then visited for each value of interest, by starting from a node that represents it. At each step, the equation represented by a visited node is stored inside the current subsets of equations and the node is disabled. All the nodes associated to equations belonging to the set of linearly dependent equations of the selected one are disabled. Disabled nodes cannot be visited again. The visit ends when all the nodes are disabled. The behavior of saboteurs are preserved by this step, since the visit always selects one equation for each branch (whether describing its potential or flow). Then, the differential equations belonging to the acquired subset are discretized by using state of art techniques of numerical differentiation and integration. Let us suppose that all derivative operators are discretized using the backward difference formula and that the access functions are replaced by symbols. If we apply the discretization with a

```
void process(const double & V_in) {
    // Evaluate the values of interest.
    V_out_gnd = 0.546961 * V_in -0.77348 * V_n1_out_ddt;
    V_n1_out  = 0.011049 * V_in +0.99447 * V_n1_out_ddt;
    // Update auxiliary variables.
    V_n1_out_ddt = V_n1_out;
}
```

Listing II: Abstracted Operational amplifier written in C++.

simulation step h to the current-voltage relation of the equation shown on line 9 of Listing I, we obtain:

$$\texttt{I\_n1\_out} \mathrel{<+} ((\texttt{V\_n1\_out} - \texttt{V\_n1\_out\_ddt})/\texttt{h}) * \texttt{c1}; \quad (1)$$

where `I_n1_out` and `V_n1_out` are respectively the current flowing through and the voltage across the capacitor at time t. While `V_n1_out_ddt` is the voltage across it at $t - h$. All the values that were argument of a time-dependent function (*e.g.*, `V_n1_out`) are also considered as values of interest. They are required in order to evaluate at each simulation step the differentials and integrals *auxiliary* variables introduced by the discretization (*e.g.*, `V_n1_out_ddt`).

3) Equation system symbolic solving: Each discretized system of equations has to be symbolically solved by considering as unknowns the values of interest. This operation requires a symbolic solver, *i.e.*, an engine able to manipulate mathematical expressions. However, before solving the system of equation, all the symbols referring to the components variables are replaced by their numerical values (*e.g.*, `r1` with `400` in Listing I). The solution of the system is a set of *linearly independent equations* each of which has the following form:

$$lhs = \sum_i (a_i * c_i) + \sum_j (i_j * c_j) \quad (2)$$

where lhs is a value of interest, c_i and c_j are numerical constants, a_i is the i-th auxiliary variable and i_j is the j-th input of the model. Listing II shows the C++ code generated by applying the abstraction flow to the running example with the voltage between nodes `out` and `gnd` as value of interest.

C. Testing framework building

A complete testing framework is built by using the previously solved fault-free and the mutated set of equations. Such a framework has the main role of orchestrator and is composed of two main functions: one executing the fault-free description and the other which executes the faulty one. The latter has two main tasks: *1)* selects one of the faulty set of equations and *2)* checks if the simulation time is inside the activation window of the selected fault. If it is outside that window, the fault-free set of equations is used instead. The two orchestrator's functions are wrapped by a testing environment which: *1)* Generates the input stimuli, *2)* executes the two functions and provides the outputs of interest to an comparator (See Section III-D) and *3)* finally updates all the auxiliary variables based on the discretization technique. If applied to the Equation 1, the update phase assigns the value `V_n1_out` to `V_n1_out_ddt`.

D. Fault simulation

The final phase starts the fault simulation and checks the presence of faults. At the end of each simulation step the outputs of interest are tested by means of a comparator function. Such a function evaluates the absolute value of the instantaneous difference between the response of the fault-free and faulty descriptions. Then, the value is compared with a *threshold* provided by the designer. If the value exceeds the threshold then the simulation is dropped and the fault is considered as detected. In conclusion, the proposed generation flow generates a C++ code which implements a complete testing framework able to simulate all modeled faults.

IV. Experimental Results

The presented testing framework generation flow has been implemented in the ADVANTAGE tool, *i.e.*, the *ADVanced ANalog Testing frAmework GEneration flow*. It exploits the tool HIFSuite [13] for parsing and manipulating the input descriptions. Verilog-AMS models have been simulated by using ELDO provided by Questa ADMS. All the experiments have taken place on a 64-bit machine running Linux with 16 GB of memory and Intel i7-3770 @ 3.40GHz.

A. Testing framework generation flow validation

The testing framework generation flow has been validated by using a set of nine benchmarks taken from an online repository of Verilog-AMS models[1]. Table I reports the features of each benchmark that are: the number of equations, inputs, outputs, internal nodes, lines of code of the Verilog-AMS description and the time required to generate its testing framework. The generation time depends mainly on the number of ddt and idt that have been discretized and on the number of injected faults. The former increases the number of auxiliary variables and thus the complexity of the expressions contained inside the final model. The latter increases the number of faulty systems of equations that have to be solved. The negligible generation times reported in table highlight the efficiency of the proposed flow even with the increasing complexity of the models. The normalized root-mean-square-deviation (NRMSD) between the response of the Verilog-AMS circuit level and the generated functional models has been evaluated in order to measure the accuracy of the models. In the worst case the NRMSD is 10^{-5}. Table II reports the simulation time required by Questa-ADMS and the proposed framework for simulating a fault-free description, a faulty version with one active fault and the fault injection campaign. The overhead caused by the mutation of the code is reported for the single active fault scenario. The simulation has been performed for *1 second* of simulated time with a fixed timestep of *5 ns*. Table III reports instead the results with a timestep which has been automatically set by Questa-ADMS to *100 ns* during the simulation. The same timestep has been used with the C++ testing framework. All the faulty simulations do not drop the simulation if a fault is detected. A sequence of values generated from a sinusoidal function has been used as testbench in both the simulation environments. The generated framework is able to achieve from two to three orders of magnitude of speed-up, thus proving to be a valuable solution for the dependability evaluation of an analog device.

B. Testing analysis

The availability of a so efficient functional testing framework allows to perform some testing analyses that would be computationally unfeasible by using a circuit level simulator. The results of a threshold sensibility analysis performed by using the testing framework of the *Accellerometer* are shown in Table IV. It reports: the type, number and percentage of

[1] www.designers-guide.org

Table I: Benchmarks features and framework generation time.

Benchmark	Equations	Inputs	Outputs	Internal Nodes	LoC	Generation Time (s)
RC1	2	1	1	1	17	**0.01**
IN2	3	2	1	2	21	**0.01**
PIFilter	4	1	1	2	21	**0.01**
IN3	5	3	1	3	31	**0.02**
Op-Amplifier	6	1	1	3	31	**0.01**
RC5	10	1	1	5	42	**0.03**
RC10	20	1	1	10	67	**0.17**
RC20	40	1	1	20	117	**1.45**
Accelerometer	66	10	8	25	123	**0.30**

Table II: Execution times of fault-free, single injected and fault campaign benchmarks with a timestep of 5 ns.

	Verilog-AMS				Testing Framework – C++					
	Fault Free	Single Active Fault		Fault Campaign	Fault Free	Single Active Fault			Fault Campaign	
Benchmark	time (s)	time (s)	over (%)	time (s)	time (s)	time (s)	over (%)	speed-up (x)	time (s)	speed-up (x)
RC1	5,046.72	5,487.12	8.73	5,523.44	4.87	5.91	21.36	**928.45**	5.87	**940.96**
IN2	5,028.54	5,828.67	15.91	11,561.44	4.46	5.16	15.70	**1,129.59**	10.12	**1,142.43**
PIFilter	5,081.49	6,133.54	20.70	12,340.74	5.10	6.31	23.73	**972.03**	12.91	**955.91**
IN3	5,312.51	6,239.96	17.46	18,713.28	4.89	5.75	17.59	**1,085.21**	17.21	**1,087.35**
Op-Amplifier	5,741.43	6,978.23	21.54	21,159.44	4.93	5.83	18.26	**1,196.95**	17.81	**1,188.06**
RC5	5,763.95	7,408.76	28.54	37,168.75	6.56	9.28	41.46	**798.36**	46.78	**794.54**
RC10	6,704.85	9,588.87	43.01	96,220.30	13.12	23.33	77.82	**411.01**	236.83	**406.28**
RC20	8,512.29	17,453.71	105.04	330,286.14	54.31	101.78	87.41	**171.48**	1,879.30	**175.75**
Accelerometer	11,917.28	18,517.34	55.38	462,407.91	47.72	67.95	42.39	**272.51**	1,584.71	**291.79**

Table III: Execution times of fault-free, single injected and fault campaign benchmarks with a timestep of 100 ns.

	Verilog-AMS				Testing Framework – C++					
	Fault Free	Single Active Fault		Fault Campaign	Fault Free	Single Active Fault			Fault Campaign	
Benchmark	time (s)	time (s)	over (%)	time (s)	time (s)	time (s)	over (%)	speed-up (x)	time (s)	speed-up (x)
RC1	316.78	397.56	25.50	403.35	0.31	0.37	19.35	**1,074.49**	0.37	**1,090.14**
IN2	313.74	404.92	29.06	812.59	0.29	0.33	13.79	**1,227.03**	0.66	**1,231.20**
PIFilter	334.19	433.36	29.67	871.91	0.35	0.43	22.86	**1,007.81**	0.87	**1,002.20**
IN3	327.15	422.98	29.29	1,274.07	0.29	0.34	17.24	**1,244.06**	1.02	**1,249.09**
Op-Amplifier	350.19	465.91	33.04	1,406.22	0.32	0.37	15.63	**1,259.22**	1.09	**1,290.11**
RC5	338.14	482.23	42.61	2,444.11	0.45	0.63	40.00	**765.44**	3.19	**766.18**
RC10	392.37	581.21	48.13	5,865.78	0.84	1.46	73.81	**398.09**	15.88	**369.38**
RC20	486.64	985.59	102.53	19,323.62	3.31	6.06	83.08	**162.64**	118.11	**163.61**
Accelerometer	538.35	901.98	67.55	22,652.32	2.24	3.14	40.18	**287.25**	78.88	**287.17**

detected faults, the detection threshold and the time required to generate and simulate the framework. The framework has been simulation for *1 second* of simulation time with a timestep of *100 ns*. For these experiments the *fault dropping feature* has been used. For the purpose ofthis experiment the current pulse with a variable amplitude (in Ampere) has been used. The aim of this experiment is to find the maximum threshold that allows to detect all the faults. Table V reports the threshold sensibility analysis performed by using the testing framework of *RC20*. In this case the analysis has been extended to a wider range of faults including: a current pulse and both short and open circuits. The results of both the tables highlight the efficiency of both the generation flow and generated testing framework with different types of faults, allowing to perform an effective testing campaign. Thanks to the flexibility of the injection process, it was possible to inject different kinds of fault inside the more complex proposed test case (*i.e.*, RC20).

V. Concluding Remarks

A fault testing methodology based on the abstraction of analog and mixed-signal models has been described in details. The employed abstraction technique produces a signal flow representation that preserves the faulty behaviors. The speed-up of some orders of magnitude is an enabling factor for performing extensive fault campaigns which normally would require a large amount of time. The flexibility of the injection flows presented in this paper allows to apply it to a wide range of fault models. Future work will extend the flow to automatically generate also the set of input stimuli.

Table IV: Threshold sensibility analysis on the Accelerometer.

Type of Fault	Number of Faults	Detected	Detection Threshold	Generation Time (s)	Simulation Time (s)
Pulse (100 nA)	25	100%	9%	0.27	22.94
Pulse (250 nA)	25	100%	11%	0.27	23.93
Pulse (300 nA)	25	100%	13%	0.26	24.20
Pulse (500 nA)	25	100%	20%	0.28	24.39
Pulse (1 uA)	25	100%	43%	0.26	24.10
Total	125			1.34	119.56

References

[1] Y. Kiliç, M. Zwoliński, and M. Z. Nski, "Behavioral Fault Modeling and Simulation Using VHDL-AMS to Speed-Up Analog Fault Simulation," *Journal of AICSP 2004*, vol. 39, no. 2, pp. 177–190.

[2] T. Gao, Y. Sun, and G. Zhao, "Analog Circuit Fault Simulation Based on Saber," in *Proc. of ICCIS 2010*. IEEE, pp. 388–391.

[3] M. Lajolo, M. Rebaudengo, M. S. Reorda, M. Violante, and L. Lavagno, "Behavioral-level test vector generation for system-on-chip designs," in *Proc. of HLDVT 2000*. IEEE Comput. Soc, pp. 21–26.

[4] H. C. Hong, "A static linear behavior analog fault model for switched-capacitor circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 4, pp. 597–609, 2012.

[5] E. Fraccaroli, M. Lora, S. Vinco, D. Quaglia, and F. Fummi, "Integration of mixed-signal components into virtual platforms for holistic simulation of smart systems," in *Proc. of DATE 2016*. IEEE, pp. 1586–1591.

[6] H. Madeira, M. Z. Rela, F. Moreira, and J. a. G. Silva, "Rifle: A general purpose pin-level fault injector," in *Proc. of EDCC 1994*, ser. EDCC-1. London, UK, UK: Springer-Verlag, pp. 199–216.

[7] R. Leveugle and A. Ammari, "Early SEU Fault Injection in Digital, Analog and Mixed Signal Circuits: A Global Flow," in *Proc. of DATE 2004*, vol. 1. IEEE Comput. Soc, pp. 590–595.

[8] L. Scheffer, L. Lavagno, and G. Martin, *EDA for IC Implementation, Circuit Design, and Process Technology*. CRC Taylor & Francis, 2006.

[9] R. Harvey, A. Richardson, E. Bruls, and K. Baker, "Analogue fault simulation based on layout dependent fault models," in *Proc. of International Test Conference*. Int. Test Conference, 1994, pp. 641–649.

[10] C. Y. Pan and K. T. Cheng, "Fault macromodeling for analog/mixed-signal circuits," in *Test Conference, 1997. Proceedings., International*. Int. Test Conference, 1997, pp. 913–922.

[11] G. C. Messenger, "Collection of Charge on Junction Nodes from Ion Tracks," *IEEE Transactions on Nuclear Science*, vol. 29, no. 6, pp. 2024–2031, 1982.

[12] Accellera Systems Initiative, "Verilog-AMS Language Reference Manual." [Online]. Available: accellera.org/downloads/standards/v-ams

[13] N. Bombieri, M. Ferrari, F. Fummi *et al.*, "HIFSuite: tools for HDL code conversion and manipulation," *EURASIP Journal on Embedded Systems*, pp. 1–20, 2010.

Table V: Test results of a fault campaign on the RC20.

Type of Fault	Number of Faults	Detected	Detection Threshold	Generation Time (s)	Simulation Time (s)
Pulse (10 mA)	20	100%	5%	1.36	30.87
Pulse (10 mA)	20	95%	10%	1.31	33.62
Pulse (10 mA)	20	90%	15%	1.34	35.90
Short Circuit	20	95%	5%	1.41	31.31
Short Circuit	20	95%	10%	1.41	31.56
Short Circuit	20	90%	15%	1.41	34.12
Open Circuit	20	100%	Any	1.44	31.95
Total	140			9.68	229.33