Fault Analysis in Analog Circuits through Language Manipulation and Abstraction

Enrico Fraccaroli*, Francesco Stefanni†, Franco Fummi* and Mark Zwolinski‡

*Dept. of Computer Science, University of Verona, Verona, Italy
†EDALAB s.r.l., Verona, Italy
‡ECS, University of Southampton, Southampton SO17 1BJ, UK

***Abstract*—Each year automotive systems are becoming smarter thanks to their enhancement with sensing, actuation and computation features. The recent advancements in the field of autonomous driving have increased even more the complexity of the electronic components used to provide such services. ISO 26262 represents the natural response to the growing concerns in terms of the functional safety of electrical safety-related systems in this area. However, if the functional safety analysis of digital devices is quite a stable methodology, the same analysis for analog components is still in its infancy. This paper aims to explore the problem of fault analysis in analog circuits and how it can be integrated into the design processes with minimum effort. The methodology is based on analog language manipulation, analog fault instrumentation and automatic abstraction. An efficient and comprehensive flow for performing such an activity is proposed and applied to complex case studies.**

***Index Terms*—Fault analysis, fault injection, analog circuit, language manipulation, abstraction**

I. Introduction

Recent trends in the global automotive industry are pointing towards cars with integrated smart features. Such "smartness" is achieved by integrating analog components, like sensors (*e.g.*, position, pressure, oxygen levels, *etc.*) and actuators (*e.g.*, steering, break, *etc.*), inside already developed digital Intellectual Property (IP) blocks. However, this process requires extra effort in order to ensure that the whole system is able to provide its functionality even in presence of faults. As a natural response, all the development phases of Original Equipment Manufacturers (OEMs) now integrate processes which allow compliance with the functional safety guidelines defined inside the ISO 26262 standard [1]. Right after the publication of the standard, several works investigated its effects on the existing approaches and how it should be applied in practice [2]–[4]. Yet if the functional safety analysis of digital devices is quite a stable methodology, the same analysis for analog components is still in its infancy [5].
The contributions of this work are:

- A study of how to model and inject common analog faults through the *manipulation* of descriptions written with different writing styles and abstraction levels.
- The application of an *abstraction* process which simplifies the manipulated descriptions, while preserving their behaviors, in order to improve simulation efficiency.

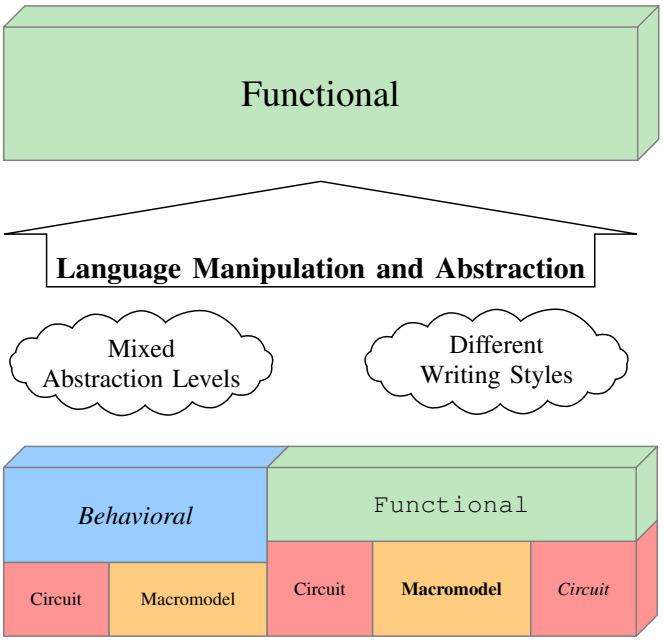

Fig. 1: Methodology overview.

Figure 1 illustrates the problems that this work addresses. As such, the target of this work is a model described at different abstraction levels (*i.e.*, circuit, macromodel, behavioral, functional) and different styles and languages.

One of the reasons of this work is the rapidly increasing complexity of analog circuits [6]. In the past this same issue was addressed by transforming an analog design from the circuit level to higher levels of abstraction, in particular to macromodel, behavioral and functional levels. If on the one hand higher levels allow us to greatly improve simulation efficiency, on the other hand they remove the connection between behavior and the actual circuit structure [7]. Injection techniques which rely on the circuit's structure cannot be applied at all the abstraction levels. It is worth noting that most of today's analog circuit designs are described at circuit level, behavioral level or a combination of the two [8].

To enhance the re-usability of digital IPs, new languages have been developed as extensions of widely used Hardware Description Languages (HDLs), *e.g.*, VHDL-AMS and

Verilog-AMS [9]. Such languages allow the concurrent design of digital and analog processes. Furthermore, designers are allowed to split complex designs into submodules and to describe each of them with a different abstraction level and writing style. An example can be found in [8] of how this feature is used. Most of the fault analysis techniques work under the single fault assumption, thus injection procedures need to inject faults only inside one submodule at a time. As a consequence, designers can selectively describe those submodules where the fault resides at circuit level and exploit the efficiency of behavioral modeling for the others. However, this freedom leads to a high heterogeneity of abstraction levels that could be simultaneously used in one design.

Efficient mixed-level simulation is needed in order to verify most mixed-signal systems [10]. This is due to the extensive use of top-down methodologies with large complex mixed-signal systems. Such flows rely on high level simulation for most of the development process, until the initial architectural design is partitioned into detailed blocks at the lowest level. It is clear that sooner or later mixed-level simulation is required in such a process. Thus, a flexible fault injection methodology is of utmost importance and should be able to inject faults in each block, regardless the abstraction level or writing style which has been used.

Based on such issues this work presents:

- a taxonomy of designs and analog fault models based on their abstraction level;
- analog faults injection inside Verilog-AMS code through automatic manipulation; and
- effective fault analysis through an efficient simulation.

Three case studies are used to better explain the application of the methodology proposed by this paper[1]. The *first case study* is a voltage-dependent resistor; its Verilog-AMS code is shown in Listing 1. The *second case study* given in Listing 2 is a controlled voltage source with hysteresis. The *third case study* shown in Listing 3 is the code of an accelerometer's interface. In this design, the variables from `V1` to `V5` are parameters of the design, while the other elements of the expressions are voltages and currents of electrical nodes. Unlike the first design, the second and third designs have no structural representations.

The paper is organized as follows. Section II contains an analysis of the abstraction levels at which an analog design can be described and the analog fault models that can be found in the literature. For each fault model, the code manipulations required to inject it are shown and explained. Such an analysis allows to reason about how to automatically model and simulate widely used analog fault models. Section III presents the abstraction flow used to transform a design to the *functional* level and explains how injected faults are preserved by the abstraction. The correlation between analog faults and designs is shown with the support of case studies in Section IV. Section V shows the simulation statistics for the generated code examples. Conclusions are drawn in Section VI.

[1]Downloaded from The Designer's Guide (www.designers-guide.org).

Listing 1: A voltage-dependent resistor.

```
module Varistor(p, n);
    inout p, n;
    electrical p, n, pi, mid;
    branch (p,pi)   br_rseries;
    branch (pi,mid) br_lseries;
    branch (mid,n)  br_cparallel;
    branch (mid,n)  br_nonlin;
    parameter real R    = 100u    from [0:inf);
    parameter real T    = 1.0     from (0:inf);
    parameter real C    = 1.0e-12 from (0:inf);
    parameter real L    = 1.0e-9  from (0:inf);
    parameter real B1   = 1.0     from (0:inf);
    parameter real B2   = 1.0     from (0:inf);
    parameter real B3   = 0.0     from (-inf:inf);
    parameter real B4   = 0.0     from (0:inf);
    parameter real Imax = 1.0e7   from (0:inf);
    parameter real Imin = 1.0e-7  from (0:inf);
    analog function real powlogV;
        input logibr, B1, B2, B3, B4;
        real  logibr, B1, B2, B3, B4;
        powlogV = pow(10.0, B1 + B2*(logibr) +
                              B3*exp(-logibr) +
                              B4*exp(logibr));
    endfunction
    analog begin : the_module
        real ibranch, logibr, vbranch, rlin;
        V(br_rseries)   <+ R * I(br_rseries);
        V(br_lseries)   <+ L * ddt(I(br_lseries));
        I(br_cparallel) <+ C * ddt(V(br_cparallel));
        // Nonlinear Branch
        ibranch = I(mid, n);
        if (ibranch > Imin) begin
            logibr = log(ibranch);
            vbranch = powlogV(logibr,B1,B2,B3,B4);
        end else if (ibranch < -Imin) begin
            logibr = log(-ibranch);
            vbranch = -powlogV(logibr,B1,B2,B3,B4);
        end else begin
            // linear interpolation for -Imin < I < Imin
            logibr = log(Imin);
            rlin = powlogV(logibr,B1,B2,B3,B4)/Imin;
            vbranch = rlin * ibranch;
        end
        V(br_nonlin) <+ T * vbranch;
    end
endmodule
```

Listing 2: A controlled voltage source with hysteresis.

```
module Vhys(in,out);
    inout  in, out;
    Voltage in, out;
    parameter real K    = 40.0;
    parameter real Vhys =  0.1;
    parameter real Vio  =  0.0;
    parameter real Vol  = -9.0;
    parameter real Voh  =  9.0;
    real Vin, Vout, offset;
    analog function real fcube;
        input x, L, H;
        real  x, L, H, a;
        begin
            a   = x / (H - L) / 1.5 + 0.5;
            fcube = (a < 0) ? L : (a > 1) ? H :
                    L + (H - L) * (3 - 2 * a) * a * a;
        end
    endfunction
    analog begin
        @(initial_step) offset = Vio + Vhys;
        // Get the input voltage.
        Vin   = V(in);
        // Evaluate and set the output voltage.
        Vout  = fcube(K * (Vin - offset), Vol, Voh);
        if (Vout == Vol) offset = Vio + abs(Vhys);
        if (Vout == Voh) offset = Vio - abs(Vhys);
        V(out) <+ Vout;
    end
endmodule
```

Listing 3: The electrical interface of an accelerometer.

```
1  input  AX, AY, AZ, TX, TY, TZ;
2  output C1, C2, C3, C4;
3  analog begin
4    I(AXD)<+ - V5  * V(AXD) + V6  * ddt(V(AX));
5    I(AYD)<+ - V7  * V(AYD) + V8  * ddt(V(AY));
6    I(AZD)<+ - V9  * V(AZD) + V10 * ddt(V(AZ));
7    I(R0D)<+ + V1  * V(R0D) + V2  * ddt(V(R0));
8    I(R1D)<+ - V3  * V(R1D) + V4  * ddt(V(R1));
9    I(R0) <+ + V23 * V(R0)  - V24 * V(R1)
10             + V25 * V(TX)  - V26 * V(TY)  + V27 * V(TZ)
11             + V28 * V(AXD) + V29 * V(AYD) - V30 * V(AZD)
12             + V31 * ddt(V(R0D)) - V32 * ddt(V(R1D));
13   I(R1) <+ - V33 * V(R0)  + V34 * V(R1)
14             - V35 * V(TX)  - V36 * V(TY)  - V37 * V(TZ)
15             + V38 * V(AXD) - V39 * V(AYD) - V40 * V(AZD)
16             - V41 * ddt(V(R0D)) + V42 * ddt(V(R1D));
17   // Outputs
18   V(C1) <+ + V11 * V(R0)  + V12 * V(R1)  + V13;
19   V(C2) <+ + V14 * V(R0)  + V15 * V(R1)  + V16;
20   V(C3) <+ + V17 * V(R0)  + V18 * V(R1)  + V19;
21   V(C4) <+ + V20 * V(R0)  + V21 * V(R1)  + V22;
22 end
```

II. Fault Taxonomy and Code Manipulation

In this section, we collect analog fault models from the literature, and categorize them into a multi-level taxonomy to show how they are applied (if possible) at each level of abstraction. The paper focuses on the following types of faults: *short-circuit/bridge*, *open-circuit*, *potential/flow pulses* and *parametric* faults. The first three models are called *saboteurs* and are usually injected by means of parametrized analog blocks. Parametric faults, on the other hand, are called *mutants* and produce small deviations or mutations of component values, and for this reason they are the hardest to detect. Table I shows a taxonomy of the aforementioned fault models based on the four analog abstraction levels proposed in [7].

In detail, a *circuit* description is usually defined by connecting SPICE primitives and must abide by the laws of conservation of energy. A *macromodel* description is a simplified circuit made of controlled sources which cannot be associated with an actual circuit but which satisfy the laws of conservation of energy. A *behavioral* description is a mathematical description with no internal structure and which satisfies the laws of conservation of energy but only at its external pins. A *functional* model is a mathematical signal flow description, which has no internal structure and which does not abide by the conservation laws.

TABLE I: Proposed taxonomy of analog fault models at different levels of abstraction.

Abstraction Level	**Fault Model**			
	Bridge/Short Circuit	Open Circuit	Potential/Flow Pulse	Parametric
Functional			✓	✓
Behavioral			✓	✓
Macromodel	✓	✓	✓	✓
Circuit	✓	✓	✓	✓

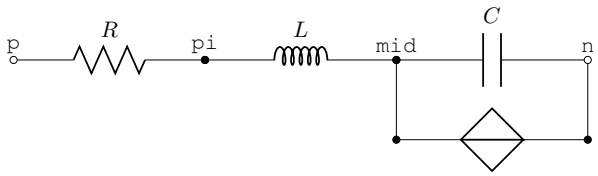

Fig. 2: Electrical topology of the voltage-dependent resistor.

A. Design taxonomy

Before moving to the categorization of the fault models, let us see how analog and mixed-signal designs can be described at each level of abstraction. Let us start with Listing 1, an example of a design described at the *circuit* level. It represents a variable resistor and, referring to the code, it comprises three basic passive elements from lines 27 to 29, *i.e.*, a resistor, an inductor and a capacitor. It also contains a controlled voltage source at line 44. The topology shown in Figure 2 can be inferred from the code of Listing 1, in particular by analyzing the differential equations defined inside the `analog begin` block. In Verilog-AMS whenever an access function is used, in this case `V` or `I`, an edge is defined inside the circuit. The existence of a topology allows us to determine at which abstraction level the design is described.

The design shown in Listing 2 represents a positive hysteresis generator controlled by the potential difference at node `in`. This is an example of a *behavioral* description and as such the energy conservation laws are satisfied only at the terminals, *i.e.*, at `in` and `out`. Internally the model has no structure; no contribution is made between pairs of analog nodes. The design shown in Listing 3 represents the electrical interface of an accelerometer, which is part of a project with the goal of designing an automotive system. This type of design is another example of a *behavioral* description, but what differs with respect to the code of Listing 2 is the writing style. Such a design is not easily injected with faults by means of methodologies that rely on the netlist and schematic of the Device Under Test (DUT).

The abstraction process developed in this work and presented in Section III generates a *functional* description. It produces code written in C++ that has no internal structure and has no analog pins that have to abide by the laws of the conservation of energy.

B. Fault taxonomy

1) Short/Bridge: The first type of fault we are going to analyze is the *short*, usually caused by two bare wires in a circuit which touch each other. Besides the different configurations that a *short* can assume, it can be considered a special case of a *bridge* fault with a zero resistance value. *Bridge* faults usually assume values which range from 0Ω to 10kΩ [11], [12]. Given the example of Figure 2, there is a total of ten *bridge* faults that can be placed inside the design: four from the electrical nodes to ground and six between all the pairs of nodes of the circuit. This is the estimated number of faults without taking into account the different resistance

Listing 4: Modified voltage-dependent resistor.

```
parameter real Ropen = 1M   from [0:inf);
analog begin : the_module
    real ibranch, logibr, vbranch, rlin;
    V(br_rseries)    <+ R * I(br_rseries);
    V(br_lseries)    <+ L * ddt(I(br_lseries));
    I(br_cparallel) <+ C * ddt(V(br_cparallel));
    // Inject OpenCircuit    at pi_mid
    V(pi, mid) <+ I(pi, mid) * Ropen;
    // Nonlinear Branch
    ...
end
```

values that each *bridge* could assume. This type of fault can be used only at the circuit and macromodel levels since it requires knowledge about the circuit topology of the design. With a good insight about the injected model, it is possible to represent such a fault at the behavioral level [13], however, since it is a design-dependent fault it cannot be automated or generalized for other designs. For injection at circuit and macromodel levels, a *short* can be modeled as a resistor. The following example represent a 1Ω *short* fault injected between node `p` and node `pi` of Listing 1:

```
I(p, pi)    <+ V(p, pi) / 1;
```

While, in the following there is a 5kΩ *bridge* fault injected between the same nodes:

```
I(p, pi)    <+ V(p, pi) / 5e03;
```

2) Open Circuit: Open fault models are saboteurs generated by missing contacts or cracks on the interconnections of a circuit. But contrary to what one might think, *opens* do not completely block the current flowing through an edge of the circuit, rather, they dramatically increase the resistance of it [14]. This fault model can be used only at the circuit and macromodel levels, for the same reasons as *short* circuits. This fault can be injected by adding a high resistance in series with an existing edge of the circuit. The following is an example of a 1MΩ *open* fault injected in series with `br_lseries` which is an edge between `pi` and `mid` of Listing 1:

```
V(br_lseries) <+ I(br_lseries) * 1e06;
```

Listing 4 shows the modified Verilog-AMS code with the aforementioned *open* fault injected.

3) Potential/Flow Pulses: Potential and flow pulses are components commonly used to inject the class of saboteurs modeling Single Event Transients (SETs). This kind of fault is physically generated by alpha particles or neutrons hitting a sensitive node of the circuit. An example of this fault described with VHDL-AMS can be found in [15], while an example described in Verilog-A can be found in [16]. Such a fault can be modeled by a controlled potential/flow source and can be injected inside Listing 1 with the following equation:

```
I(pi) <+ Pulse;
```

where *Pulse* is a value controlled by the simulation environment and updated at each simulation step to match the waveform of the desired fault model (*e.g.*, double exponential, damped sinewave).

4) Parametric: Parametric faults can have many causes, *e.g.* extra residues of metal during manufacturing of an Integrated Circuit (IC) or even deterioration due to aging. We can consider *parametric* faults to be all those that make the design parameters fall outside acceptable boundaries. It is infeasible to test all the possible variations of the values of such parameters since they belong to the domain of real numbers and, thus, can assume an infinite number of values. As a consequence, many works which try to deal with this type of faults have developed techniques which aim at reducing the number of faults that have to be tested [17], [18]. Let us take the code of Listing 1 and in particular the declaration of the parameters `R`, `L` and `C`. Injection of faulty parameters can be can be performed with two types of code manipulations. The first is directly inside the design, by changing the default values associated with the parameters as follows:

```
// Before      R = 100u
parameter real R = 125u     from [0:inf);
// Before      L = 1.0e-9
parameter real L = 4.8e-9   from (0:inf);
// Before      C = 1.0e-12
parameter real C = 9.4e-12  from (0:inf);
```

However, if the design is instantiated inside a hierarchical model, parametric faults can be injected by modifying the instantiation as follows:

```
varistor #(.R(125u), .L(4.8e-9), .C(9.4e-12))
    varistor_instance(p, n);
```

Either way, the proposed flow is also compatible with techniques reported in the literature aiming at reducing the number of analyzed faults.

All the methodologies mentioned in this section have some limitations. Some are able to perform fault injection only at a specific abstraction level, while others allow only the injection of a single type of fault. Furthermore, most of them are not automatic or cannot be automated. For this reason, a generalized and automatic procedure is required, as it is already the case in the digital world. But above all, a fault model is required, which can be used at every level of abstraction and following different writing styles. The proposed methodology unifies the way analog faults are injected and disregarding the initial abstraction level of a design, it produces a unique functional-level description with all injected faults.

III. Code Abstraction with Faults

The code abstraction process transforms the Verilog-AMS code written at different levels of abstraction to a functional description. Such a manipulation allows an efficient fault analysis to be performed by reducing the simulation complexity of an Analog and Mixed-Signal (AMS) description. Figure 3 exemplifies the steps which implement the fault analysis flow. It starts from a preliminary analysis of the code which acquires the structure of the design, in particular the lists of nodes and edges. The topology is used to automatically generate the list of fault locations based on the types of fault models which need to be injected. A new faulty description is generated for each fault location by injecting the equations describing the fault model associated with the location.

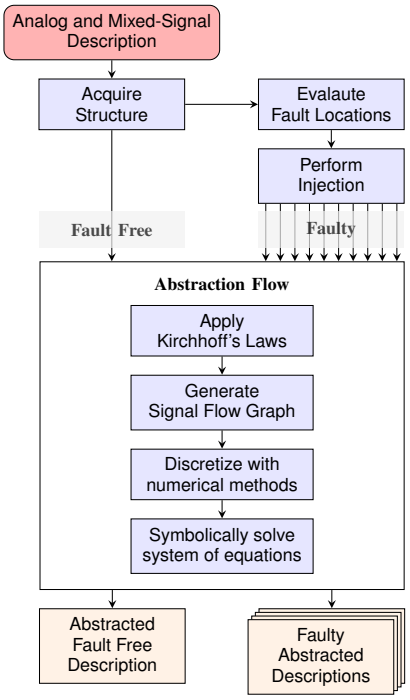

Fig. 3: Overview of the fault analysis flow.

A. Abstraction flow

First, the abstraction flow performs a nodal and loop analysis by applying Kirchhoff's laws at the nodes and edges of the circuit. The new equations produced by the analysis are combined with those defined inside the analog model. Then, a Signal Flow Graph (SFG) describing the input-output behavior of the system is generated. It associates equations to nodes and the relations between them as branches. It contains one equation (either potential or flow) for each edge of the circuit. Once the graph is completed, the derivatives and integrals functions contained in its equations are discretized by means of numerical differentiation and integration techniques. This process replaces these functions with symbols (*i.e.*, variables) which are evaluated during the simulation according to the selected discretization technique. The system of discretized equations is then solved using a symbolic computation library. For this work we used the C++ library, GiNaC [19]. Such a flow moves the complexity of solving the system of equations from runtime to generation time.

An abstraction flow similar to the one used in this paper has been proposed in [20]. Here we use a similar process, applying it to the fault-free and faulty models and then by re-combining the resulting simplified models. However, the aim here is to manipulate the code in such a way that any abstraction methodology, with a similar flow, can easily handle the injected faults. Another similar approach was proposed [21], where Nagi, Chatterjee and Abraham discussed a fault simulator called DRAFTS. That work exploits the performance of high level simulation by abstracting the analog circuit to the behavioral level and then transforming it from the continuous s-domain of Laplace to the discrete z-domain. At the end of the abstraction process, they perform the fault simulation in the discretized z-domain. The work in [22] is an example of a high-level testing framework which combines abstraction and fault injection in the digital domain. Unlike the work outlined here, it first abstracts the model and then injects abstracted digital faults.

B. Preserving faulty behaviors

As shown in Figure 3, the abstraction flow is applied to both the fault-free design and all the faulty variants. It is thus imperative that the abstraction does not discard the information concerning the injected faults (*i.e.*, parameters or the entire set of equations). For the purpose of fault analysis it is vital that faulty behaviors are preserved by the process of abstraction.

Concerning *catastrophic faults*, their injection is done by adding new equations to the existing ones, as shown in Section II-B. Each new equation represents a new edge of the circuit, as such its presence is considered during the generation of Kirchhoff's equations. When the abstraction generates the SFG, the graph inevitably contains an equation describing either the potential or the flow of the injected edge. The same applies to *parametric faults*. These are manipulations of components parameters which are contained inside the equations describing the behavior of the component. As a consequence, at least one of the two equations (*i.e.*, the ones describing the flow or the potential) associated with the component is selected and thus its presence is preserved.

IV. Fault analysis flow

Once the abstraction flow has been applied to all the fault-free and faulty descriptions they are combined inside the same model. As such, the output of the flow is a C++ class which contains both the fault-free description and all the faulty ones. This allows us to to improve the simulation efficiency.

An analog fault can impact on different edges of the circuit and vice versa. Thus the equations evaluating the potential and flow of such edges change based on the currently active fault. An efficient mechanism which allows switching between the faulty and fault-free equations associated with an edge must be implemented. Since the final code is written in C++, the natural way to do this is with a `switch` statement.

Listing 5 shows the result of the injection on the design of Listing 1. As mentioned above, a switch statement is used to change the equations describing the potential and flow associated with an edge depending on the currently active fault, identified by the variable `faultSelector`. For sake of readability only few cases of the switch have been shown. While building these switch statements, an analysis is carried out to determine if there are cases with the same behavior. This means that the these faults produce the same mutated potential and flow equations. This allows an insight of how a fault injected in one place of the circuit impacts on the physical values in other places. Furthermore, it allows us to determine equivalence between faults.

Listing 5: Injected voltage-dependent resistor.

```
void Varistor::analog_process() {
  double ibranch, logibr, vbranch, rlin;
  // Impact of fault at p_pi
  switch (faultSelector) {
    case 1:  // Inject OpenCircuit at p_pi
    case 5:  // Inject OpenCircuit at pi_mid
    case 9:  // Inject OpenCircuit at mid_n
    I_p_pi = I_pi_mid_ddt*V1 + V_mid_n_ddt*V2 + V_p*V3;
    V_p_pi = I_pi_mid_ddt*V4 + V_mid_n_ddt*V5 + V_p*V6;
    break;
    case 2:  // Inject ShortCircuit at p_pi
    I_p_pi = I_pi_mid_ddt*V7 + V_mid_n_ddt*V8 + V_p*V9;
    V_p_pi = I_pi_mid_ddt*V10+ V_mid_n_ddt*V11+ V_p*V12;
    break;
    ...
    default: // Fault Free
    I_p_pi = I_pi_mid_ddt*V29+ V_mid_n_ddt*V30+ V_p*V31;
    V_p_pi = I_pi_mid_ddt*V32+ V_mid_n_ddt*V33+ V_p*V34;
    break;
  }
  // Impact of fault at pi_mid
  switch (faultSelector) { ... }
  // Impact of fault at mid_n
  switch (faultSelector) { ... }
  // Nonlinear Branch
  ibranch = I_mid_n;
  if (ibranch > Imax) {
    logibr  = log(ibranch);
    vbranch = powlogV(logibr, B1, B2, B3, B4);
  }
  else if (ibranch < -Imin) {
    logibr  = log(-ibranch);
    vbranch = -powlogV(logibr, B1, B2, B3, B4);
  }
  else {
    // linear interpolation for -Imin < I < Imin
    logibr  = log(Imin);
    rlin  = powlogV(logibr, B1, B2, B3, B4) / Imin;
    vbranch = ibranch * rlin;
  }
  V_mid_n_1 = vbranch;
  // Updating variables
  I_pi_mid_ddt = I_pi_mid;
  V_mid_n_ddt  = V_mid_n;
}
```

Listing 6: Injected positive voltage hysteresis.

```
double Vhys::fcube( double x, double L, double H ) {
  double a = x / (H - L) / 1.5 + 0.5;
  return (a<0)?L:(a>1)?H:(L+(H-L)*(3-2*a)*a*a);
}
void Vhys::initial_step() {
  offset = Vio + Vhys;
}
void Vhys::analog_process() {
  // Inject Potential Pulse at in
  V_in += potential_pulse_01
  Vin = V_in;
  Vout = fcube(K * (Vin - offset), Vol, Voh);
  if (Vout == Vol) offset = Vio + abs(Vhys);
  if (Vout == Voh) offset = Vio - abs(Vhys);
  // Inject Potential Pulse at out
  V_out = Vout + potential_pulse_02;
}
```

Listing 6 shows the result of the injection on the design shown in Listing 2. As mentioned in Section II-A, this is a behavioral description which lacks a structure and thus it can be injected only with potential pulses. As a consequence, no switch statement is required. However, a unique variable called `pulse` is generated for each injected fault (*i.e.*, two in this case). This allows control of the activation of each fault by modifying the value of its `pulse` variable. Even if the number of catastrophic faults is limited in this case, the design presents a number of different parameters that the proposed flow can modify with parametric faults.

Listing 7: Injected accelerometer's interface.

```
void Accelerometer::analog_process() {
  // Inject Potential Pulse at R0
  V_R0 = V_AX * V1 + V_AX_ddt * V2 +
         V_AY * V3 + V_AY_ddt * V4 +
         V_AZ * V5 + V_AZ_ddt * V6 +
         V_R0_ddt * V7  + V_R0D_ddt * V8  +
         V_R1_ddt * V9  + V_R1D_ddt * V10 +
         potential_pulse_01 * V11;
  // Inject Potential Pulse at R1
  V_R1 = V_AX * V12 + V_AX_ddt * V13 +
         V_AY * V14 + V_AY_ddt * V15 +
         V_AZ * V16 + V_AZ_ddt * V17 +
         V_R0_ddt * V18 + V_R0D_ddt * V19 +
         V_R1_ddt * V20 + V_R1D_ddt * V21 +
         potential_pulse_02 * V22;
  ...
  // Inject Potential Pulse at C4
  V_C4 = V_AX * V23 + V_AX_ddt * V24 +
         V_AY * V25 + V_AY_ddt * V26 +
         V_AZ * V27 + V_AZ_ddt * V28 +
         V_R0_ddt * V29 + V_R0D_ddt * V30 +
         V_R1_ddt * V31 + V_R1D_ddt * V32 +
         potential_pulse_10 * V33;
  // Updating variables
  V_AX_ddt  = AX; V_AY_ddt = AY; V_AZ_ddt = AZ;
  V_R0_ddt  = V_R0;  V_R1_ddt  = V_R1;
  V_R0D_ddt = V_R0D; V_R1D_ddt = V_R1D;
}
```

Listing 7 shows a piece of the injected code of the accelerometer's interface of Listing 3. Such a code presents the same features of the voltage-dependent resistor, thus the only injectable faults where pulses.

V. Experimental Results

This section presents the results of the application of the manipulation and abstraction flow to the three case studies presented in Section I. The experiments have been performed on a 64-bit Linux machine, equipped with 8 GB of memory and a CPU with two 2.70GHz cores. All the steps and manipulations presented in this paper have been implemented in an automatic tool which makes use of the APIs provided by a commercial tool [23]. These APIs provide only the *front-end* to parse Verilog-AMS code and a *back-end* to produce C++ codes. The generated C++ code has been compiled with gcc 4.8.5. Simulation of the original Verilog-AMS code has been performed with a leading commercial SPICE-based simulator.

Table II shows the injection statistics for the three types of fault presented in Section II-B. It reports the number of *open* circuits, *short* circuits, *potential* pulses, *flow* pulses and the total number of injected faults. Table II also reports the time required to simulate the fault-free original code, the fault-free abstracted code and the time required to perform the entire fault campaign. All the three scenarios have been executed with a step of 1 *us* for 1 *second* of *simulated time*.

Final results shown that performing an entire fault campaign with the proposed flow takes much less time than simulating

TABLE II: Injection statistics for the proposed case studies.

	Injection Statistics					Simulation Statistics		
						Verilog-AMS	C++	
Benchmark	Open	Short	Pot. Pulse	Flow Pulse	**Total**	Fault Free	Fault Free	Fault Campaign
Accelerometer Interface	0	0	10	0	10	341.03 *s*	0.07 *s*	0.87 *s*
Varistor	3	10	6	6	25	256.16 *s*	0.11 *s*	3.05 *s*
Voltage Hysteresis	0	0	2	0	2	244.93 *s*	0.05 *s*	0.11 *s*

the original code. Even though this paper does not directly address the problem of parametric fault injection, such high simulation performances can be used to test a great number of deviations of designs parameters, thus also improving the efficiency of parameter fault analysis.

VI. Concluding remarks

This paper presents a methodology to manipulate and abstract analog circuit models in order to perform an efficient fault analysis. An overview is given of the main problems that designers and also tools have to face to perform fault analysis. The heterogeneity of modeling styles and abstraction levels, with which an analog model can be described, complicates even further the process of fault injection. To shed some light on this, a taxonomy of the analog fault models that can be found in literature is presented in conjunction with the fault model that can be applied at each level of abstraction.

An abstraction flow is also presented aiming at speeding up the simulation of the injected description. This is achieved by transforming the description from any abstraction level to the functional level. It is also shown that this process of abstraction does not remove or nullify the effects of injected faults. Then, the entire flow is applied to a series of case studies described at different abstraction levels and with different writing styles. The simulation results show the validity of the presented work even with a heterogeneous set of designs.

Extensive analog fault analysis becomes feasible in this way. The results confirm that the effectiveness of fault models which can be found in the literature is tightly dependent on the modeling style and abstraction level at which the design is written. In future work, we will study a generalized fault model which can be effectively injected into any type of design.

References

[1] ISO, "ISO/DIS 26262 - Road vehicles – Functional safety," Geneva, Switzerland, Tech. Rep., July 2011. [Online]. Available: https://www.iso.org/standard/43464.html

[2] M. Hillenbrand, M. Heinz, K. D. MÃijller-Glaser, N. Adler, J. Matheis, and C. Reichmann, "An approach for rapidly adapting the demands of iso/dis 26262 to electric/electronic architecture modeling," in *Proceedings of 21st IEEE International Symposium on Rapid System Protyping*, June 2010, pp. 1–7.

[3] M. Born, J. Favaro, and O. Kath, "Application of ISO DIS 26262 in practice," *Proceedings of the 1st Workshop on Critical Automotive applications Robustness & Safety - CARS '10*, p. 3, 2010.

[4] R. Rana, M. Staron, C. Berger, J. Hansson, M. Nilsson, and F. Törner, "Increasing efficiency of iso 26262 verification and validation by combining fault injection and mutation testing with model based development." pp. 251–257, 2013.

[5] S. Sunter, "Closing the loop between analog design and test," *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 894–897, 2016.

[6] S. Sunter, K. Jurga, P. Dingenen, and R. Vanhooren, "Practical random sampling of potential defects for analog fault simulation," *Proceedings - International Test Conference*, vol. 2015-Febru, pp. 1–10, 2015.

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

[8] S. Abughannam, L. Wu, W. Mueller, C. Scheytt, W. Ecker, and C. Novello, "Fault Injection and Mixed-Level Simulation for Analog Circuits - A Case Study," in *ANALOG 2016; 15. ITG/GMM-Symposium*, 2016, pp. 1–6.

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

[10] H. Kundert, Ken and Chang, "Top-Down Design and Verification of Mixed-Signal Circuits," pp. 1–8, 2005.

[11] H. T. Vierhaus, W. Meyer, and U. Glaser, "CMOS bridges and resistive transistor faults: IDDQ versus delay effects," in *Proceedings of IEEE International Test Conference - (ITC)*, Oct 1993, pp. 83–91.

[12] J. M. Acken, "Special Applications of the Voting Model for Bridging Faults," *IEEE Journal of Solid-State Circuits*, vol. 29, no. 3, pp. 263–270, 1994.

[13] Y. J. Chang, C. L. Lee, J. E. Chen, and C. Su, "Behavior-level fault model for the closed-loop operational amplifier," *Journal of Information Science and Engineering*, vol. 16, no. 5, pp. 751–766, 2000.

[14] C. Henderson, J. Soden, and C. Hawkins, "The Behavior and Testing Implications of Cmos Ic Logic Gate Open Circuits," *1991, Proceedings. International Test Conference*, no. 1, pp. 302–310, 1991.

[15] 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.

[16] S. N. Ahmadian and S. G. Miremadi, "Fault injection in mixed-signal environment using behavioral fault modeling in Verilog-A," in *Proc. of BMAS 2010*. IEEE, sep, pp. 69–74.

[17] M. J. Barragan, H. G. Stratigopoulos, S. Mir, H. Le-Gall, N. Bhargava, and A. Bal, "Practical simulation flow for evaluating analog/mixed-signal test techniques," *IEEE Design & Test*, vol. 33, no. 6, pp. 46–54, Dec 2016.

[18] A. Singhee and R. A. Rutenbar, "Statistical blockade: Very fast statistical simulation and modeling of rare circuit events and its application to memory design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 8, pp. 1176–1189, 2009.

[19] C. Bauer, A. Frink, and R. Kreckel, "Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language," *Elsevier JSC*, vol. 33, no. 1, pp. 1 – 12, 2002.

[20] M. Lora, S. Vinco, E. Fraccaroli, D. Quaglia, and F. Fummi, "Analog models manipulation for effective integration in smart system virtual platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

[21] N. Nagi, A. Chatterjee, and J. a. Abraham, "Fault simulation of linear analog circuits," *Analog Integrated Circuits and Signal Processing*, vol. 4, no. 3, pp. 245–260, nov 1993.

[22] A. Fin and F. Fummi, *LAERTE++: An Object Oriented High-Level TPG for SystemC Designs*. Boston, MA: Springer US, 2004, pp. 105–117.

[23] N. Bombieri, M. Ferrari, F. Fummi *et al.*, "HIFSuite: tools for HDL code conversion and manipulation," *EURASIP JES*, vol. 2010, no. 1, pp. 1–20, 2010.