Enhancement of Fault Injection Techniques Based on the Modification of VHDL Code

Juan-Carlos Baraza, Joaquín Gracia, Sara Blanc, Daniel Gil, and Pedro-J. Gil, *Member, IEEE*

***Abstract*—Deep submicrometer devices are expected to be increasingly sensitive to physical faults. For this reason, fault-tolerance mechanisms are more and more required in VLSI circuits. So, validating their dependability is a prior concern in the design process. Fault injection techniques based on the use of hardware description languages offer important advantages with regard to other techniques. First, as this type of techniques can be applied during the design phase of the system, they permit reducing the time-to-market. Second, they present high controllability and reachability. Among the different techniques, those based on the use of saboteurs and mutants are especially attractive due to their high fault modeling capability. However, implementing automatically these techniques in a fault injection tool is difficult. Especially complex are the insertion of saboteurs and the generation of mutants. In this paper, we present new proposals to implement saboteurs and mutants for models in VHDL which are easy-to-automate, and whose philosophy can be generalized to other hardware description languages.**

***Index Terms*—Dependability validation, fault tolerance, hardware description languages (HDLs), logic design, mutants, physical faults, saboteurs, VHDL-based fault injection, VLSI.**

I. Introduction

THE NEW DEEP submicrometer technologies are increasingly sensitive to physical faults, both to those due to external phenomena (i.e., transient faults such as single event upsets (SEUs), single event transient (SETs), etc.) and to internal defects (i.e., intermittent and permanent faults). Moreover, this sensitivity implies not only a raise of the fault rate, but also an increment of the likelihood of appearing multiple faults [1]–[3]. For this reason, the dependability of systems must be analyzed. This analysis can be either the study of the incidence of faults on the system (called *error syndrome* analysis) or checking the design specifications (called *validation*). The objective of the error syndrome analysis is to detect those parts of the system which are most sensitive to faults, and eventually, to choose the most suitable fault-tolerance mechanisms (FTMs). The aim of the validation is to verify that the system and/or its built-in FTMs accomplish the design specifications in presence of faults.

If the dependability is analyzed at early phases of the design cycle, both time and money can be saved in the development process. A common experimental method to validate the dependability of a fault tolerant system (FTS) is fault injection, which is defined in [4] as *the deliberate introduction of faults into a system (the target system)*.

Fault injection techniques can be classified in three main categories [5]: physical [also known as Hardware Implemented Fault Injection (HWIFI)], software implemented (SWIFI), and simulation-based. HWIFI is accomplished at physical level, disturbing the hardware with parameters of the environment (heavy ions radiation, electromagnetic interferences, etc.) or modifying the logic value of the pins of the integrated circuits. The objective of SWIFI consists of reproducing at software level the errors that would have been produced upon occurring faults in the hardware or the software. In simulation-based fault injection, the system under test is simulated in another computer system. The faults are induced altering the logical values during the simulation.

Simulation-based fault injection is a useful experimental way to evaluate the dependability of a system during the design phase, thus reducing the time-to-market [6]–[8]. Another interesting advantage of this group of techniques with regard to others is that those based on simulation offer both high observability and controllability of all the modeled components [9].

Particularly, there exist a group of fault injection techniques based on the use of hardware description languages (or HDLs) as modeling languages. The most popular high-level HDLs are VHDL, Verilog, and SystemC. In our case, we work with VHDL [10].

These techniques are widely applied, due to the advantages of employing an HDL. The present work is framed in this group of techniques. Fig. 1 shows a classification of VHDL-based fault injection techniques. Nevertheless, both this taxonomy and the description of the injection techniques can be generalized to any other HDL.

Simulator commands technique is based on the use of simulator commands to modify the value or timing of the model signals and variables, without altering the VHDL code [6]. In the remaining techniques, the original VHDL code of the model is modified, either inserting *saboteurs* [6], [11], [12] or mutating the components of the model [6], [13], [14].

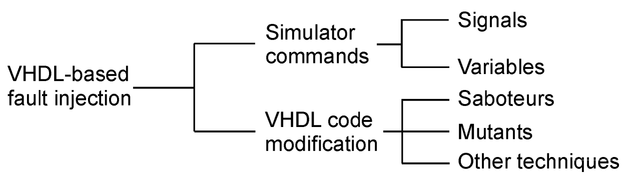


Fig. 1. VHDL-based fault injection techniques.

Manuscript received September 1, 2006; revised July 17, 2007. This work was supported in part by the Spanish research projects "Improvement of VHDL-based fault injection techniques" (TIC2002-02491) and "Estudio de las técnicas de inyección de fallos en modelos en VHDL: Emulación de fallos en FPGA y uso de nuevas técnicas de simulación distribuida" (TEC2005-05119).

The authors are with the Grupo de Sistemas Tolerantes a Fallos (GSTF) of the Departamento de Informática de Sistemas y Computadores (DISCA), Universidad Politécnica de Valencia (UPV), 46022 Valencia, Spain (e-mail: jcbaraza@disca.upv.es; jgracia@disca.upv.es; dgil@disca.upv.es; pgil@disca.upv.es).

Digital Object Identifier 10.1109/TVLSI.2008.2000254

The techniques labeled as *Other techniques* are implemented by extending the VHDL language, either by adding new data types and signals, or modifying the VHDL resolution functions [7], [15]. The new data types and signals defined include the fault behavior description. However, these techniques require developing *ad hoc* compilers and simulators, and introducing control algorithms to manage the language extensions.

There are works related to fault injection with saboteurs and mutants in other areas like test or field-programmable gate array (FPGA)-based fault emulation, although the objective of the study in each area is quite different.

In dependability analysis, the objective can be either to verify the sensitivity to physical faults or validate the effectiveness of the FTMs of a simulation model (not necessarily synthesizable) of the system under analysis, by modifying the operation of the model at simulation time.

In test, the aim of fault injection is to accelerate the test process by obtaining reduced test pattern lists injecting faults at higher abstraction levels, like register-transfer (RT) or system. For instance, in [16], a fault simulation tool has been developed for system models designed in Verilog at RT level. The purpose of this tool is not only to verify the model, but also to get the test pattern set that obtains the best correlation in the fault coverage between RT level and gate level. The RT fault simulator is based on simulating a modified version of the system model in which a number of zero-delay buffers (similar to *serial simple saboteurs*—see Section IV-A) are strategically inserted according to two statistical criteria: optimistic and pessimistic analysis. The modified model is then simulated with a commercial fault simulation tool called Verifault. Also, in [17], a fault simulation tool is developed, but, in this case, it accepts VHDL models at system level. Another important difference with the work in [16] is that the fault simulator developed performs the fault simulation by itself, instead of using a commercial fault simulator. Last, but not least, another important dissimilarity is the fact that the original model is *mutated* by inserting a special type of function able to alter the behavior of the system (see Section II-C for details). Finally, in [18], a technique to obtain the stratified coverage of a complex (that is, composed of multiple internal components) Verilog model at RT level is presented. Like in [16], Mao and Gulati use a gate-level commercial fault simulator (in this case, Verifault-XL) to simulate a modified version of the model in which a number of zero-delay buffers are judiciously inserted.

In FPGA-based fault emulation, the objective of fault injection by using saboteurs and mutants is to synthesize into an FPGA a modified version of the original model that can be managed externally in order to emulate a faulty behavior. Interesting works in this area are [19], where mutants are implemented, [20] that applies saboteurs, and [21] that implements behavioral saboteurs.

On the other hand, our research group has developed VFIT [22], [23], a VHDL-based fault injection tool that applies several of the previously described techniques. In fact, only the *other techniques* group has not been implemented due to their excessive complexity.

More information about dependability analysis and fault injection can be found in [5].

In this paper, we intend to explain the drawbacks of some models of saboteurs and mutants existing in the literature [24], to justify the introduction of new implementations. Some models of saboteurs and mutants will be discussed and revised, and new models will be proposed. Also, we will show how these new designs can be automatically inserted in a model in order to perform a fault injection campaign, illustrating the description of every proposal with an application example. To confirm the effectiveness of the enhancements introduced, we also include the results of a set of injection experiments in which we compare aspects such as the duration of the simulation and analysis phases (i.e., the temporal cost of injecting the faults and comparing the faulty simulation trace to the fault-free one), or the model size, and of course, the overall data extracted from the injection experiment.

The distribution of this paper is as follows. In Section II, we make a short review of the most common VHDL-based fault injection techniques. Section III describes the fault injection environment summarily. In Section IV, the models of saboteurs developed are discussed and a new set of models are proposed. Section V analyzes the models of mutants currently used and presents a new implementation method. Results of implementing the new methods proposed are shown in Section VI. Finally, both a discussion of the results and a proposal of future work are provided in Section VII.

II. VHDL-Based Fault Injection Techniques

A. Fault Injection Using Simulator Commands

This fault injection technique is based on using the commands of the simulator at simulation time, in order to modify the value or timing of the signals and variables of the model [24]. Moreover, as VHDL generic constants are managed as special variables, it is possible to inject some non-usual fault models, such as delay faults [8].

Using simulator commands it is possible to inject transient, permanent, and intermittent faults. Though, there exists one restriction: due to the special nature of variables in VHDL, it is not possible to inject permanent faults in variables.

This technique is the easiest one to implement and its temporal cost (to perform the simulation) is by far the lowest. However, the number of fault models that can be injected is smaller than with the other techniques.

B. Fault Injection With Saboteurs

A saboteur is a special VHDL component added to the original model [8], [12]. When activated, the mission of this component is to alter the value, or timing characteristics, of one or more signals, simulating the occurrence of a fault. During the normal operation of the system, instead, the component remains inactive. Saboteurs affect to the ports of the components in the model. Thus, this technique is applicable only to structural descriptions.

Attending to how saboteurs are inserted in the model, two types can be distinguished: serial and parallel [6]. As Fig. 2(a) shows, a serial saboteur interposes between a component input port (*I* in the figure) and its source signal (*O* in the figure), whereas a parallel saboteur [see Fig. 2(b)] is added as an additional source (*S* in the figure) of a given signal.

Parallel saboteurs have two important drawbacks respect to serial. First, implementing them is noticeably more complex,

TABLE I
FAULT MODELS INJECTED BY VFIT AT GATE AND RT LEVELS

Injection technique	Transient faults	Permanent/Intermittent faults
Simulator commands	Pulse[a], Bit-flip[b], Indetermination, Delay	Stuck-at (0,1), Indetermination, Open-line, Delay
Saboteurs	Pulse[a], Bit-flip[b], Indetermination, Delay	Stuck-at (0,1), Indetermination, Open-line, Delay, Short, Bridging, Stuck-open
Mutants	Stuck-then, Stuck-else, Assignment control, Dead process, Dead clause, Micro-operation, Local stuck-data, Global stuck-data	Stuck-then, Stuck-else, Assignment control, Dead process, Dead clause, Micro-operation, Local stuck-data, Global stuck-data

[a]Applied to combinational logic, it represents a Single Event Transient (SET)
[b]Applied to storage elements (registers and memory), it represents a Single Event Upset (SEU)

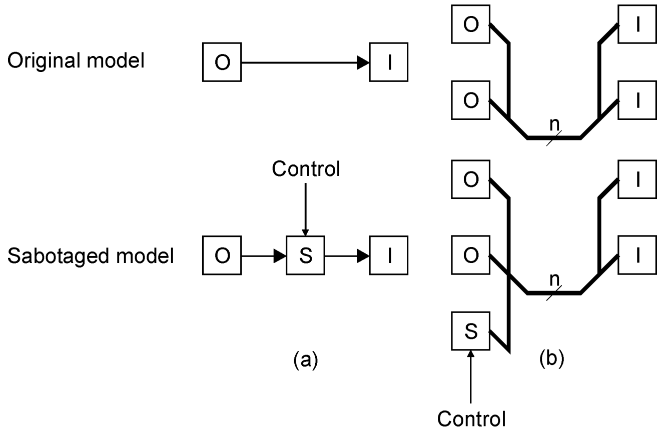


Fig. 2. Types of saboteurs. (a) Serial. (b) Parallel.

because it is necessary to modify the data type of the signal affected, as well as the resolution function associated to the data type (a resolution function defines how values from multiple sources are resolved into a single value). Second, they allow to inject fewer fault models. For these reasons, their implementation has no special interest. So, in this paper, only serial saboteurs will be considered.

C. Fault Injection With Mutants

A mutant is a component that replaces another component. While inactive, it works like the original component, but when it is activated, it behaves like the component in presence of faults. The mutation can be made in three ways:

- by adding saboteurs to structural model descriptions;
- by modifying structural descriptions replacing sub-components (i.e., a NAND gate can be replaced by a NOR gate);
- by modifying syntactical structures of behavioral descriptions.

There can exist lots of possible mutations in a VHDL model, so representative subsets of faults at logical and RT levels must be considered [13], [14], [25]: replacing the values of conditions in if and case statements (called *stuck-then*, *stuck-else*, *dead clause*, etc.), disturbing assignment statements (*assignment control*, *global stuck-data*, etc.), disturbing operators in expressions (*micro-operation*, *local stuck-data*), etc.

In our case, we have considered the following fault models [22].

- *Stuck-Then*: Replacement of the if condition by true.
- *Stuck-Else*: Replacement of the if condition by false.
- *Assignment Control*: Disturbing an assignment operation.
- *Dead Process*: Elimination of the sensitivity list of a process.
- *Dead Clause*: Elimination of a clause in a case.
- *Micro-Operation*: Disturbing an operator.
- *Local Stuck-Data*: Disturbing the value of a variable, constant, or signal in an expression.
- *Global Stuck-Data*: Elimination of all value modifications of a variable or signal in an architecture.

Many of these fault models do not have a direct correspondence with physical faults, but they can show somehow an erroneous internal operation.

III. FAULT INJECTION ENVIRONMENT

The GSTF has developed a fault injection tool called VFIT (VHDL-based fault injection tool) [22], [23], that runs on PC computers (or compatible) under Windows and is model-independent. Although it admits models at any abstraction level, it has been mainly used on models at gate and RT levels.

With VFIT it is possible to inject faults automatically applying the simulator commands technique. It is also feasible to inject faults using saboteurs and mutants, but in this case, the injection process needs the intervention of the user because the insertion of the saboteurs and the generation of mutants are not automatic.

It can inject permanent, transient, and intermittent faults. When applied to models at gate and RT levels, it uses a wide set of fault models that try to be representative of deep submicrometer technologies (see Table I). This set surpasses the classical stuck-at (for permanent faults) and bit-flip (for transient faults).

The experiment configuration is carried out through VFIT's graphic user interface (GUI). Among other functions, this GUI allows the user (with the help of a built-in VHDL parser) to select a list of fault targets among all the possible targets in the model. The class of the fault targets eligible depends directly on the fault injection technique applied (i.e., model signals and variables for simulator commands; inputs and internal connection signals of the model components for saboteurs; and special VHDL sentences for mutants). Also, for each fault target (and again depending on the fault injection technique applied), a number of fault models suitable to inject into it can be selected.

Later, an *injection scheduler* "decides" that at a given time instant (we call it the "injection instant"), the value of one or several points of the system (the fault targets) must behave in a wrong way, only once for a short time (simulating the occurrence of a transient fault), for a short time but repeatedly (simulating the occurrence of an intermittent fault), or permanently (until the end of the simulation). At simulation time, the *injection manager* runs the simulator indicating these parameters. What "wrong behavior" of the fault targets means depends strongly on the injection technique used.

- In the case of simulator commands, the injection consists on modifying directly the internal value or timing of the fault target(s) by using the commands of a simulation language (in our case, Tcl).
- When saboteurs are used, the injection consists on modifying directly (also by using simulator commands) the control lines that manage one or several saboteurs inserted in the original model. In this way, the saboteur(s) activated will propagate the affected lines with erroneous values or timing.
- When injecting faults with mutants, the injection is very similar to the injection with saboteurs. By means of simulator commands, an erroneous sentence will be "executed" instead of the correct one.

During the simulation phase, VFIT automatically selects randomly a fault target from the list, and then, a particular fault model to inject on it.

The output of an injection experiment can be either an error syndrome analysis or a validation. In both cases, output data are a set of tables. In case of an error syndrome analysis, tables contain among other values: propagation latencies, percentages of propagated errors, and percentages of failures. In the case of performing a validation, tables show propagation, detection and recovery latencies, percentages of propagated, detected, and recovered errors, detection and recovery coverages, and failure percentages.

IV. Automating the Insertion of Saboteurs

In this section, after discussing the main advantages and drawbacks of other saboteur models existing in the literature and previously developed, we describe a new set of saboteur models implemented. Also, we include an example that explains how to automate the insertion of saboteurs using the new proposal.

A. Previous Models

So far, VFIT can inject faults using serial saboteurs inserted manually in the design. The models of saboteurs implemented are as follows [24].

- *Serial Simple Saboteur (SSS):* It interrupts the connection between an unidirectional local port of a component and its formal port, modifying either its value or its timing.
- *Serial Simple Bidirectional Saboteur (SSBS):* It has two bidirectional ports and a read/write (R/W) signal that determines the direction of the perturbation.
- *Serial Complex Saboteur (SCS):* It breaks the connection between two unidirectional local ports and their formal ports, modifying either their values or their timing.
- *Serial Complex Bidirectional Saboteur (SCBS):* It has two couples of bidirectional ports and a (R/W) signal that determines the direction of the perturbation.
- *N-Bit Unidirectional Simple Saboteur (*nUSS): It applies to n-bit unidirectional buses (for instance, address and control). It has been implemented by means of a structural description, using n SSSs.

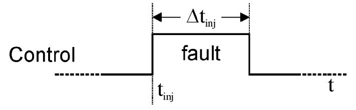


Fig. 3. Timing of fault injection.

- *N-Bit Bidirectional Simple Saboteur (*nBSS*):* It is used with n-bit bidirectional buses (for instance, data and control), and it is composed by n SSBSs.
- *N-Bit Unidirectional Complex Saboteur (*nUCS*):* It applies to n-bit unidirectional buses, and it is composed by $n/2$ SCSs.
- *N-Bit Bidirectional Complex Saboteur (*nBCS*):* It is used with n-bit bidirectional buses and composed by $n/2$ SCBSs.

Every saboteur is controlled by means of the following three inputs.

- Control, whose mission is the timing of the injection: its activation determines both the injection instant (t_{inj}) and the fault duration (Δt_{inj}). It can be seen more clearly in Fig. 3.
- Selection, that allows selecting the fault model to be injected.
- R/W, which indicates, in the bidirectional versions, the direction of the perturbation.

Although this technique requires an extra complexity due to the addition of these control signals, saboteurs allow to inject more fault models than simulator commands (see Table I). This makes the technique attractive enough.

However, at the time we intended to incorporate the models developed to VFIT, some problems were found when we tried to automate the selection of the most adequate saboteur model in each case. The main causes were the excessive number of saboteur models and the way they are implemented (by means of structural descriptions). We have tried a new set of models that fix some ambiguity difficulties, reduce the number of saboteurs, and simplify their complexity, and consequently, also the complexity of the sabotaged design.

B. Enhanced Models

The new models of saboteurs proposed, shown in Fig. 4, are as follows [26].

- *Unidirectional Serial Saboteur (USS):* It is the same model as the SSS in the previous set, although the USS allows injecting new fault models.
- *Bidirectional Serial Saboteur (BSS):* It is similar to the SSBS in the first set, but like in the previous case, the fault model set that can be injected has been extended. Also, it eliminates the R/W control signal.
- *N-Bit Unidirectional Serial Saboteur (*nUSS*):* This model replaces all the unidirectional multi-bit models in prior model set.
- *N-Bit Bidirectional Serial Saboteur (*nBSS*):* It replaces all bidirectional multi-bit models in the former proposal and eliminates the R/W control signal.

As the timing of Control and Selection inputs are identical, we have implemented an "optimized" version of these models in which the fault injection is managed only by Selection input. The idea is simple: when an injection is in progress, Selection

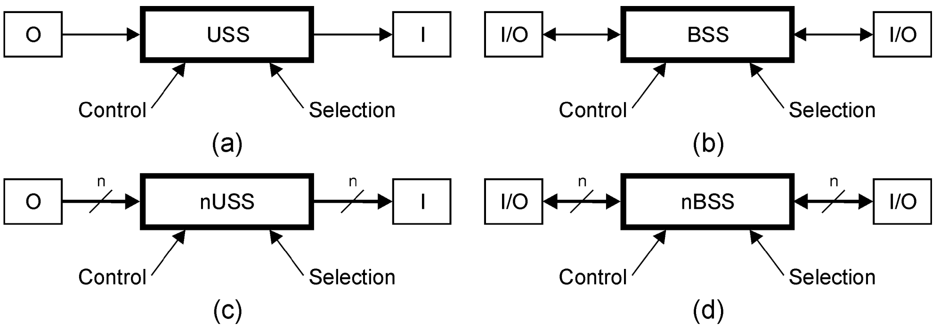


Fig. 4. New set of saboteurs implemented. (a) USS. (b) BSS. (c) nUSS. (d) nBSS [26].

indicates the fault(s) to be injected; but while no fault is injected, the value of Selection must represent a "no-fault" injection. However, this reduced version has a negative aspect: only single faults and multiple faults in the domain of time can be injected. To inject faults in the domain of space, the original scheme must be used.

As an example, we show a simplified scheme of the BSS saboteur, written in VHDL pseudo-code [24] in the following.

```
architecture behavioral of BSS
begin
  process (I, O, R/W, Control)
  begin
    if Control = '1' and not Control 'stable then
      fault_type_selection;
      if R/W = '1' then
        O <= f_inj (I,Selection);
      else
        I <= f_inj (O,Selection);
      end if;
    else
      if R/W = '1' then
        O <= I;
      else
        I <= O;
      end if;
    end if;
  end process;
end architecture;
```

This following new set of saboteur models has important differences with respect to prior ones.

- All models have been implemented using behavioral descriptions. This simplifies greatly their code and, what is more important, also the code of the design including the saboteurs. Moreover, the n-bit versions can be used for vectors of any length, because their length is defined by means of a generic parameter. Every time an n-bit saboteur is added to the model, the actual value of the generic parameter must be set.
- The number of saboteurs has been reduced to ease their automatic insertion. Now, depending on both the length (1 bit or n bits) and the mode (that is, the directionality) of the port sabotaged, only one model can be chosen.
- The bidirectional versions have the capability of injecting the fault only in the direction that data flow. In this way, the R/W input used in the models of prior version is not needed anymore, thus reducing the overhead. In the reduced version used to inject single faults, without the Control input, the spatial overhead is even more diminished.

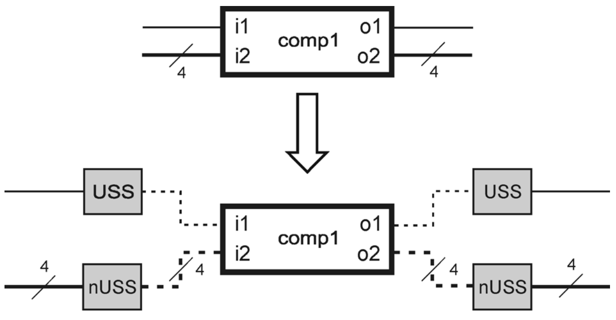


Fig. 5. Example of perturbation of a model. Distribution of saboteurs [26].

- They can inject more fault models: pulse, short, and bridging.

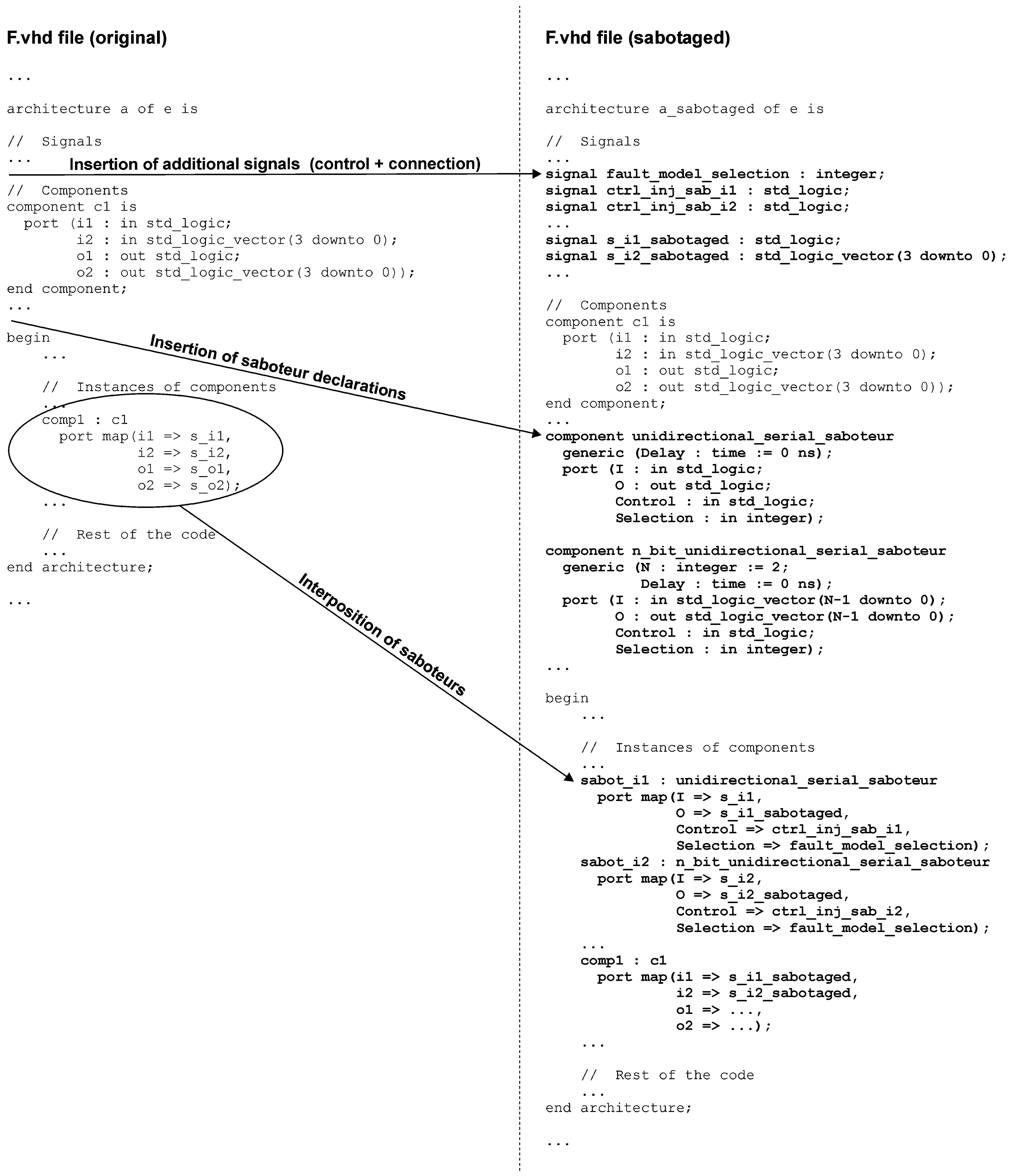
C. Automatic Insertion of Saboteurs in the Design

The task of modifying automatically a source code seems apparently very complex. However, if the injection tool includes a parser, this is not actually so. From a syntactical tree of the model containing its complete structure, it is possible to go over the tree and generate a new copy of the source files, inserting new sentences or modifying other existing as needed. The insertion of saboteurs involves the following three actions:

1) declaring the signals required to activate the saboteurs and to select the fault model to be injected;
2) declaring the components of the saboteurs introduced;
3) inserting the instances of the saboteurs, interposing between local and formal ports of the sabotaged components; this also implies declaring new signals to connect the saboteurs to local ports, and modifying the original mapping of ports.

Fig. 5 shows an example of a sabotaged model. Shaded boxes and dashed lines in lower scheme represent the saboteurs and the connection signals added to the model, respectively. To simplify the scheme, the control signals (Control and Selection) have been omitted.

Fig. 6 describes how the three actions affect to the VHDL code of the model. To simplify, only the insertion of two saboteurs is shown, but the operation is exactly the same for all of them. In Fig. 6, the original VHDL code is shown at the left side and the perturbed code at the right side. Here, the text in bold types represents the new code.


Insertion of additional signals (control + connection)
Insertion of saboteur declarations
Interposition of saboteurs

F.vhd file (original)

```
...

architecture a of e is

//  Signals
...

//  Components
component c1 is
  port (i1 : in std_logic;
        i2 : in std_logic_vector(3 downto 0);
        o1 : out std_logic;
        o2 : out std_logic_vector(3 downto 0));
end component;
...

begin
    ...

    //  Instances of components
    ...
    comp1 : c1
      port map(i1 => s_i1,
               i2 => s_i2,
               o1 => s_o1,
               o2 => s_o2);
    ...

    //  Rest of the code
    ...
end architecture;

...
```

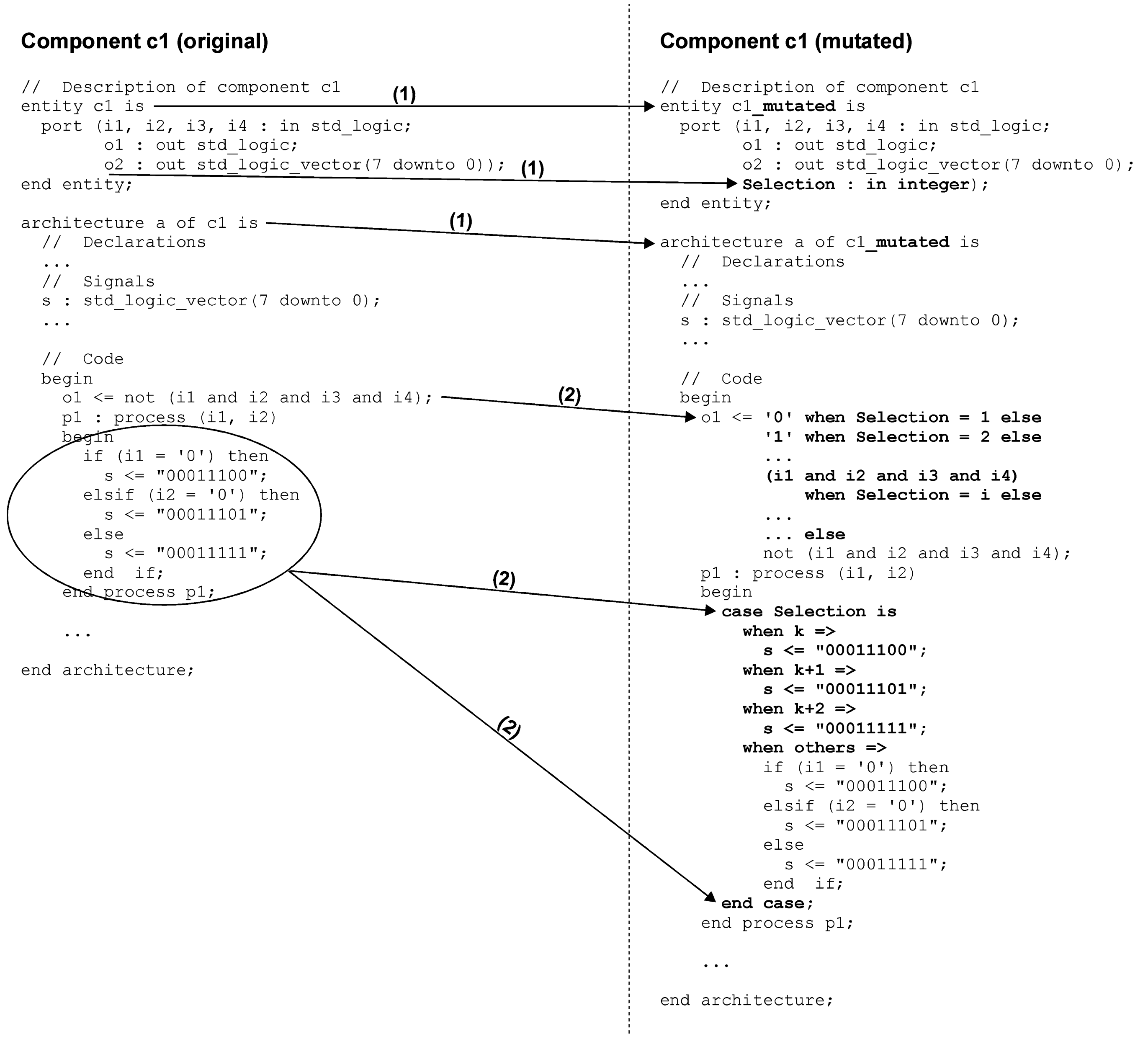
F.vhd file (sabotaged)

```
...

architecture a_sabotaged of e is

//  Signals
...
signal fault_model_selection : integer;
signal ctrl_inj_sab_i1 : std_logic;
signal ctrl_inj_sab_i2 : std_logic;
...
signal s_i1_sabotaged : std_logic;
signal s_i2_sabotaged : std_logic_vector(3 downto 0);
...

//  Components
component c1 is
  port (i1 : in std_logic;
        i2 : in std_logic_vector(3 downto 0);
        o1 : out std_logic;
        o2 : out std_logic_vector(3 downto 0));
end component;
...
component unidirectional_serial_saboteur
  generic (Delay : time := 0 ns);
  port (I : in std_logic;
        O : out std_logic;
        Control : in std_logic;
        Selection : in integer);

component n_bit_unidirectional_serial_saboteur
  generic (N : integer := 2;
           Delay : time := 0 ns);
  port (I : in std_logic_vector(N-1 downto 0);
        O : out std_logic_vector(N-1 downto 0);
        Control : in std_logic;
        Selection : in integer);
...

begin
    ...

    //  Instances of components
    ...
    sabot_i1 : unidirectional_serial_saboteur
      port map(I => s_i1,
               O => s_i1_sabotaged,
               Control => ctrl_inj_sab_i1,
               Selection => fault_model_selection);
    sabot_i2 : n_bit_unidirectional_serial_saboteur
      port map(I => s_i2,
               O => s_i2_sabotaged,
               Control => ctrl_inj_sab_i2,
               Selection => fault_model_selection);
    ...
    comp1 : c1
      port map(i1 => s_i1_sabotaged,
               i2 => s_i2_sabotaged,
               o1 => ...,
               o2 => ...);
    ...

    //  Rest of the code
    ...
end architecture;

...
```

Fig. 6. Example of perturbation of a model. Modification of the VHDL code.

It is possible to distinguish in Fig. 6 the three actions aforementioned. Remark that in the signal declaration, not only the control and selection signals are included, but also those signals required to connect the saboteurs.

With the new set of models proposed, automating the insertion of saboteurs in a model (in previously selected locations) will be relatively easy, by using a VHDL parser as VFIT does. This facility is also used in other tools [18].



```
// Description of component c1
entity c1 is
  port (i1, i2, i3, i4 : in std_logic;
        o1 : out std_logic;
        o2 : out std_logic_vector(7 downto 0));
end entity;

architecture a of c1 is
  // Declarations
  ...
  // Signals
  s : std_logic_vector(7 downto 0);
  ...

  // Code
  begin
    o1 <= not (i1 and i2 and i3 and i4);
    p1 : process (i1, i2)
    begin
      if (i1 = '0') then
        s <= "00011100";
      elsif (i2 = '0') then
        s <= "00011101";
      else
        s <= "00011111";
      end  if;
    end process p1;

    ...

end architecture;
```

```
// Description of component c1
entity c1_mutated is
  port (i1, i2, i3, i4 : in std_logic;
        o1 : out std_logic;
        o2 : out std_logic_vector(7 downto 0);
        Selection : in integer);
end entity;

architecture a of c1_mutated is
  // Declarations
  ...
  // Signals
  s : std_logic_vector(7 downto 0);
  ...

  // Code
  begin
    o1 <= '0' when Selection = 1 else
          '1' when Selection = 2 else
          ...
          (i1 and i2 and i3 and i4)
              when Selection = i else
          ...
          ... else
          not (i1 and i2 and i3 and i4);
    p1 : process (i1, i2)
    begin
      case Selection is
        when k =>
          s <= "00011100";
        when k+1 =>
          s <= "00011101";
        when k+2 =>
          s <= "00011111";
        when others =>
          if (i1 = '0') then
            s <= "00011100";
          elsif (i2 = '0') then
            s <= "00011101";
          else
            s <= "00011111";
          end  if;
      end case;
    end process p1;

    ...

end architecture;
```

Fig. 7. Example of mutation of a model. Modification of the VHDL code of a component.

V. Automating the Generation of Mutants

Injecting faults using mutants is quite more difficult than with the other two techniques described in Section II. The main problem lies on the spatial overhead introduced due to the generation of the mutations of the model. Nevertheless, in modern computers the storage is not actually a problem, so implementing this technique is nowadays more feasible.

In this section, after discussing the drawbacks of two approaches of implementation of mutants, a new method is presented. Also, an example of automatic generation is shown.

A. Previous Approaches

VFIT can inject faults using mutants inserted manually in the design (see Section III). In this subsection, the methods followed to implement this technique are described.

The first approach to implement mutant-based fault injection consists on generating multiple replicas of the architectures of all the components in the model, where every replica includes one modification (or mutation) in the VHDL code [24]. Each modification corresponds to the injection of one fault.

By means of the VHDL configuration mechanism (that is, the configuration statement), multiple versions (mutations) of the model can be generated. Also, there exists another configuration (without faults) that includes the original versions of all the model components.

The injection consists on selecting and simulating one of the multiple mutated configurations of the model. Due to the *static* nature of the configurations, only permanent faults can be injected using this approach, and moreover, from the very beginning of the simulation.

To fix this problem, a *dynamic* approach has been developed. It is based on the use of guarded signals together with the configuration mechanism [24]. In this way, at simulation time it is possible to stop the simulation of the original version of the model, and restart it simulating a faulty configuration. By using a number of simulator commands, the status of the simulation (that includes the simulation time and the value of all the signals and variables of the model) of the original version of the model can be saved on a file, and the same status is restored in the simulation of the faulty configuration. With the dynamic approach it is possible to inject (at any injection time) permanent, transient, and intermittent faults.

However, this implementation has a serious drawback: the synchronization (that is, saving and restoring the simulation status) between the simulation of the fault-free architecture and the faulty architecture involves an enormous temporal cost. In [24], a comparison of the temporal cost of the three fault injection techniques implemented in VFIT was presented. The results showed that the average simulation time (that is, the duration of simulation phase) was more than 100 times longer when using mutants than when using simulator commands, evidently due to the synchronization between the simulations.

B. New Proposal to Implement Mutants

To avoid synchronizing simulations, we suggest a “brute force” implementation. What we propose is quite simple: to generate a unique mutated version of every architecture used in the model that includes all the possibilities of mutation considered previously in the setup phase [26]. Obviously, if no statement is selected to be mutated in a particular architecture, it is not necessary to mutate the component.

In most cases, the modifications in the code are included by using if and case statements, although there are other possibilities, as shown in the example in Figs. 7 and 8. The aim of this type of modifications is to allow choosing among the correct statement and multiple wrong versions. For this purpose, a new input port (called Selection) must be inserted in the interface of the entity. The mission of Selection port is to specify the particular mutation to be activated, by asking its value in every “branch” of the mutated code. We call “branch” to every statement inserted to select between the correct operation and the wrong ones. The condition to activate one of the options is that the value of Selection coincides with the value specified in the selection statement.

Another modification required is to declare a Fault_Selection signal in the upper level of the model, which will be associated (mapped) to every local Selection port of the mutated components inserted, replacing the original ones. With this approach, also injecting faults becomes very easy. By using simulator commands, the value of Fault_Selection signal can be modified at simulation time. In this way, it is possible to inject faults of the same time characteristics than with simulator-commands technique: transient, permanent, and intermittent.

Remark that, respect to prior approaches, the new method reduces not only the temporal overhead, but also the spatial, as multiple entire replicas of each architecture are replaced by only one that includes all the modifications. However, some temporal overhead could be expected in the simulation time due to the higher complexity of the mutated model.

C. Automatic Generation of Mutants

This new proposal to implement mutants is so simple that automating the generation of mutants of a given model is not complicated at all. Assuming that an injection tool has a parser, locating in the code the target statements to be mutated and replacing them with new ones is very easy. Next, we show a practical example.

Fig. 7 represents the mutation of a component inserted in a given model. At the left side, we can see the original VHDL code of the component, and at the right side, the mutated code. Here, the text in bold types represents the modifications introduced. The arrows labeled with (1) correspond to modifications in the interface, and those labeled with (2) to the statements’ mutation. In the example, a *signal assignment* and an if statement have been mutated. The *signal assignment* has been replaced with a *conditional signal assignment*, and to mutate the if, a case statement has been inserted. Both operations are relatively easy to perform automatically.

Fig. 8 shows the modifications required in the top level of the design. As commented in Section IV, the changes introduced are of two types. On the one hand, to declare the signal that selects the mutation to activate at injection time. On the other hand, to replace the original components with the mutated ones; this affects to both the component declaration and instantiation.

VI. Comparison of the Injection Techniques Using the New Models of Saboteurs and Mutants

Notation

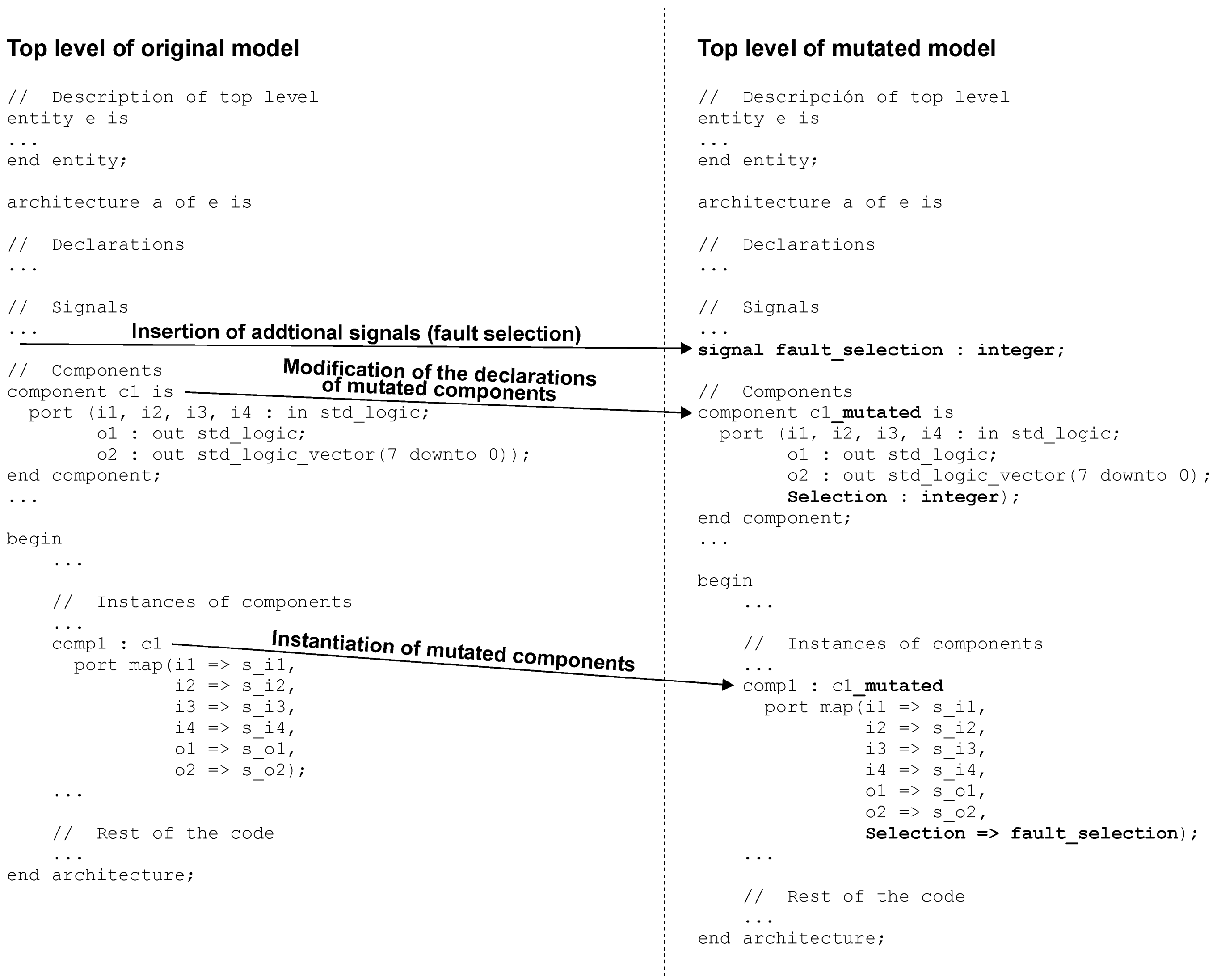
The following notation is used in the remainder of this paper:

t_{inj}	injection instant;
t_p	time when the error is activated;
t_d	time when the activated error is detected by detection mechanisms;
t_r	time when the detected error is recovered by recovery mechanisms;
N_{Injected}	number of faults injected;
$N_{\text{Activated}}$	number of activated errors;
N_{Detected}	number of errors detected by detection mechanisms;
$N_{\text{Detected\_recovered}}$	number of errors detected by detection mechanisms, and recovered by recovery mechanisms;
L_p	propagation latency;
L_d	detection latency;
L_r	recovery latency;
P_A	percentage of activated faults;
C_d	error detection coverage;
C_r	error recovery coverage.

A. Experiment Set-Up

In [24], we compared the three injection techniques applying the former models of saboteurs and mutants. In this section, we intend to repeat the experiments carried out then, and compare the results obtained to the ones in [24]. The most relevant injection parameters are the following.

1) *System Model:* A 16-bit academic fault-tolerant micro-computer system, duplex with cold stand-by sparing, parity detection and watchdog timer [22].


Insertion of addtional signals (fault selection)
Modification of the declarations of mutated components
Instantiation of mutated components

Top level of original model

```
// Description of top level
entity e is
...
end entity;

architecture a of e is

// Declarations
...

// Signals
...

// Components
component c1 is
  port (i1, i2, i3, i4 : in std_logic;
        o1 : out std_logic;
        o2 : out std_logic_vector(7 downto 0));
end component;
...

begin
    ...

    // Instances of components
    ...
    comp1 : c1
      port map(i1 => s_i1,
               i2 => s_i2,
               i3 => s_i3,
               i4 => s_i4,
               o1 => s_o1,
               o2 => s_o2);
    ...

    // Rest of the code
    ...
end architecture;
```

Top level of mutated model

```
// Descripción of top level
entity e is
...
end entity;

architecture a of e is

// Declarations
...

// Signals
...
signal fault_selection : integer;

// Components
component c1_mutated is
  port (i1, i2, i3, i4 : in std_logic;
        o1 : out std_logic;
        o2 : out std_logic_vector(7 downto 0);
        Selection : integer);
end component;
...

begin
    ...

    // Instances of components
    ...
    comp1 : c1_mutated
      port map(i1 => s_i1,
               i2 => s_i2,
               i3 => s_i3,
               i4 => s_i4,
               o1 => s_o1,
               o2 => s_o2,
               Selection => fault_selection);
    ...

    // Rest of the code
    ...
end architecture;
```

Fig. 8. Example of mutation of a model. Modification of the VHDL code of the top level.

2) *Injection Technique:* Simulator commands, saboteurs, and mutants.
3) *Number of Faults:* 3000 single faults per experiment.
4) *Fault Types and Duration:* Permanent and transient with a duration defined according to a uniform distribution function in the range $[0.1\mathrm{T_{cycle}}, 10.0\mathrm{T_{cycle}}]$, where $\mathrm{T_{cycle}}$ is the CPU clock cycle.
5) *Fault Models:* See Table I.
6) *Workload:* Calculus of an arithmetic series of n integer numbers and Bubblesort.

We have inserted the equivalent new models of saboteurs in the same places as in the original experiments. Also, we have introduced in the model all the same mutations as in the original experiments. Finally, we have performed an experiment using simulator commands technique to be used as reference in the comparisons.

In order to make a complete comparison of the injection techniques (and of the different versions), we have measured two types of data in all cases.

On the one hand, performance parameters, are like the following.

- The size of the source code. This gives an idea of the spatial overhead introduced by every technique and version.
- The average simulation time (of one 3000-fault injection experiment).
- The average analysis time (of one experiment). It is the duration of the analysis phase.
- The size of the simulation traces, distinguishing between the golden run and the average size of the faulty traces. We have included it to see if the new methods provoke a significant growth in the number of simulation events.

Also, we have compared the outcomes of the proper injection experiment. In this way, we have measured the following:

- percentage of activated errors

$$P_A = \frac{N_{\text{Activated}}}{N_{\text{Injected}}} \times 100$$

- propagation, detection, and recovery latencies

$$L_p = t_p - t_{\text{inj}} \qquad L_d = t_d - t_p \qquad L_r = t_r - t_d$$

TABLE II
COMPARISON OF THE INJECTION TECHNIQUES AND VERSIONS: SOURCE CODE SIZE

	Parameter
Injection technique (version)	Source code size[a] (code lines)
Simulator commands	1141
Saboteurs (old)	1754
Saboteurs (new)	2016
Mutants (dynamic)	251821
Mutants (new)	2283

[a]The code corresponding to common libraries is not included.

- error detection and recovery coverages

$$C_d = \frac{N_{\text{Detected}}}{N_{\text{Activated}}} \times 100 \qquad C_r = \frac{N_{\text{Detected\_recovered}}}{N_{\text{Activated}}} \times 100.$$

Respect to the original experiments, we have changed the computer where the simulations have been run. In the original experiments, the computer had a Pentium II microprocessor at 350 MHz and 192 MB of RAM. In the experiments described here, the computer used has a Pentium 4 microprocessor at 2.80 GHz and 1 GB of RAM. Also, current release of the injection tool uses enhanced injection and analysis algorithms. For these reasons, we have repeated all the original experiments in order to get a coherent comparison of the results.

Due to the number of experiments performed (up to 20, divided in four campaigns per injection technique and version, varying both the fault duration and workload run), and the high amount of data collected, it is useful to classify these following injection campaigns.

- *Campaign 1:* Injection of transient faults when the model runs the arithmetic series.
- *Campaign 2:* Injection of permanent faults when the model runs the arithmetic series.
- *Campaign 3:* Injection of transient faults when the model runs the Bubblesort algorithm.
- *Campaign 4:* Injection of permanent faults when the model runs the Bubblesort algorithm.

The results will be presented in a tabular way distinguishing the four injection campaigns carried out and with two separate groups of tables: one for the performance parameters and the other one for the analysis outcomes.

The comparison of performance parameters is indicated in Tables II–VI. Table II shows only the source code size, because this parameter is independent of the fault duration and the workload run, whereas the other tables show the remaining parameters.

In Tables VII–X, we illustrate the comparison of the analysis results.

B. Performance Comparison of Saboteur Approaches

Looking at Table II, the size of the source code of the sabotaged model when using the new saboteur models is slightly greater than when using the former ones, although the difference is not very important.

With regard to the simulation time, we can observe that there is a variation in this parameter depending on the workload. When the workload is the arithmetic series, the simulation time is lower in the new saboteur models, whereas it is lower in the old models when the Bubblesort algorithm is run. Anyway, the overall temporal cost (that is, the sum of the simulation time plus the analysis time) is lower with the new models regardless the workload run, although slightly greater than with simulator commands technique. The reason is the higher complexity of the sabotaged model.

In relation to the size of simulation traces, two aspects should be taken into consideration. First, the size of the golden run is smaller when injecting with the new saboteurs (in fact, it has the same size as with simulator commands) than with the former ones. However, the average size of injection traces is bigger with the new saboteurs, although it is smaller than with simulator commands. The size of a faulty simulation trace depends on the final effect of the fault(s) injected.

On the one hand, it can happen that no error is propagated to the system. In this case, the simulation trace will have approximately the same size as the golden run.

If any errors are propagated to the system, the activation of the FTMs implies more simulation events, and hence, the simulation trace is bigger.

Finally, if a crash occurs in the system due to the propagation of the fault(s) injected, the simulation trace is smaller than the golden run.

So, if the average size of injection traces is bigger in the new saboteurs, we can infer that the new saboteur models provoke a higher number of propagated errors, and hence, they activate more times the FTMs of the model.

Considering that the new saboteur models allow to inject a wider fault model set (as indicated in Section IV-B), and that they can be integrated more easily, we can conclude that the new set of saboteurs has improved the performance of the technique in relation to the former models.

C. Performance Comparison of Mutant Approaches

When comparing the performance of both mutant approaches, enormous differences can be found in both the system model size and the simulation time.

In effect, the size of the mutated model with the dynamic mutants is more than 100 times bigger than with the new approach. As we pointed out in Section V-B, this is because in an n-fault injection experiment, a total of n replicas of different architectures as well as n model configurations are generated. Although these components are small, when n is big enough (say 3000, like in the experiments described here) make the code to grow considerably.

With regard to the simulation time, it is between 183 and 805 times lower with the new mutants than with the dynamic approach. In fact, the cost of the new mutants is similar to the cost of the new saboteur approach.

On the other hand, no important differences have been found in the analysis phase duration.

Finally, in relation to the size of simulation traces, two details can be considered. First, in the new approach there is a reduction of the average injection trace size respect to the golden run,

TABLE III
COMPARISON OF THE INJECTION TECHNIQUES AND VERSIONS (CAMPAIGN 1): PERFORMANCE

	Parameter			
			Simulation Trace Size (kB)	
Injection technique (version)	Simulation Time (min)	Analysis Time (min)	Golden run	Injections
Simulator commands	21	5	327	310
Saboteurs (old)	24	8	336	252
Saboteurs (new)	20	6	327	252
Mutants (dynamic)	16923[a]	6	327	323
Mutants (new)	21	5	327	294

[a]i.e. 11 days, 18 hours, and 3 minutes.

TABLE IV
COMPARISON OF THE INJECTION TECHNIQUES AND VERSIONS (CAMPAIGN 2): PERFORMANCE

	Parameter			
			Simulation Trace Size (kB)	
Injection technique (version)	Simulation Time (min)	Analysis Time (min)	Golden run	Injections
Simulator commands	23	6	327	284
Saboteurs (old)	24	7	336	251
Saboteurs (new)	22	5	327	262
Mutants (dynamic)	10474[a]	5	327	327
Mutants (new)	23	5	327	292

[a]i.e. 7 days, 6 hours, and 34 minutes.

TABLE V
COMPARISON OF THE INJECTION TECHNIQUES AND VERSIONS (CAMPAIGN 3): PERFORMANCE

	Parameter			
			Simulation Trace Size (kB)	
Injection technique (version)	**Simulation Time (min)**	**Analysis Time (min)**	**Golden run**	**Injections**
Simulator commands	**68**	**30**	**1780**	**1761**
Saboteurs (old)	**76**	**66**	**1816**	**1344**
Saboteurs (new)	**89**	**32**	**1780**	**1456**
Mutants (dynamic)	**16690[a]**	**26**	**2313**	**2329**
Mutants (new)	**86**	**29**	**1780**	**1628**

[a]i.e. 11 days, 14 hours, and 10 minutes.

TABLE VI
COMPARISON OF THE INJECTION TECHNIQUES AND VERSIONS (CAMPAIGN 4): PERFORMANCE

	Parameter			
			Simulation Trace Size (kB)	
Injection technique (version)	Simulation Time (min)	Analysis Time (min)	Golden run	Injections
Simulator commands	64	29	1780	1617
Saboteurs (old)	76	62	1817	1347
Saboteurs (new)	86	30	1780	1514
Mutants (dynamic)	15388[a]	22	2313	2342
Mutants (new)	84	29	1780	1630

[a]i.e. 10 days, 16 hours, and 28 minutes.

whereas in the dynamic approach the average injection trace size is very similar to the golden run. Notice that when running the Bubblesort algorithm, all the simulation traces are bigger with the dynamic mutants than with both the new mutants and simulator commands. The reason is that in this workload there are much more simulation events. Second, there is a generalized increment in the average injection trace size in new mutants respect to simulator commands. As explained in Section VI-C, this may imply that the system FTMs are activated more times; that is, it might indicate a greater error propagation rate.

TABLE VII
COMPARISON OF THE FAULT INJECTION TECHNIQUES AND VERSIONS (CAMPAIGN 1): ANALYSIS RESULTS

	Parameter					
Injection technique (version)	P_A (%)	l_p (ns)	l_d (ns)	l_r (ns)	C_d (%)	C_r (%)
Simulator commands	63.63	900	34491	103600	41.70	31.64
Saboteurs (old)	53.73	4661	13348	50965	74.01	0.93
Saboteurs (new)	74.27	5070	24539	90937	53.32	15.31
Mutants (dynamic)	100.00	108	22608	52961	30.80	22.13
Mutants (new)	46.93	17925	37089	96053	53.69	23.30

TABLE VIII
COMPARISON OF THE FAULT INJECTION TECHNIQUES AND VERSIONS (CAMPAIGN 2): ANALYSIS RESULTS

	Parameter					
Injection technique (version)	P_A (%)	l_p (ns)	l_d (ns)	l_r (ns)	C_d (%)	C_r (%)
Simulator commands	72.83	5546	39992	119563	50.43	21.37
Saboteurs (old)	54.07	4415	12076	80010	74.60	1.11
Saboteurs (new)	67.40	5843	31477	94693	50.79	14.64
Mutants (dynamic)	100.00	132	29691	0	7.40	0.00
Mutants (new)	47.30	18978	35399	93715	55.95	23.40

TABLE IX
COMPARISON OF THE FAULT INJECTION TECHNIQUES AND VERSIONS (CAMPAIGN 3): ANALYSIS RESULTS

	Parameter					
Injection technique (version)	P_A (%)	l_p (ns)	l_d (ns)	l_r (ns)	C_d (%)	C_r (%)
Simulator commands	64.93	1210	36194	111563	42.15	39.17
Saboteurs (old)	54.37	4904	14525	3940	77.07	0.49
Saboteurs (new)	75.83	6138	27657	89372	59.38	24.66
Mutants (dynamic)	100.00	108	729	39332	95.47	86.13
Mutants (new)	48.03	20506	51594	104862	72.73	42.12

TABLE X
COMPARISON OF THE FAULT INJECTION TECHNIQUES AND VERSIONS (CAMPAIGN 4): ANALYSIS RESULTS

	Parameter					
Injection technique (version)	P_A (%)	l_p (ns)	l_d (ns)	l_r (ns)	C_d (%)	C_r (%)
Simulator commands	74.83	6659	39589	131567	55.41	34.65
Saboteurs (old)	54.53	5077	13442	3636	76.77	0.37
Saboteurs (new)	69.13	6991	33177	93038	59.11	28.88
Mutants (dynamic)	99.47	199	16795	131030	9.92	1.41
Mutants (new)	48.37	21435	51073	103326	72.02	42.32

D. Overall Comparison of Performance of Injection Techniques

Comparing only the injection techniques and considering exclusively the new methods proposed, one could assert that injecting faults with saboteurs has a higher temporal cost than with simulator commands.

In fact, the advantages of the new saboteurs are not evident because their improvements are not quantitative but qualitative: they can be automatically inserted in the model quite easily and they allow to inject a wider fault model set.

It is evident that saboteur-based technique introduces a strong overhead in the model, due to the insertion of new components (the saboteurs) and signals to connect and manage them. However, this overhead does not result in an important spatial overcost, and the temporal cost is absolutely affordable by current computers, so the technique still remains interesting.

Concerning mutants, with the new approach they have now the same complexity (both temporal and spatial) as saboteurs. The reduction of the spatial overhead can be because of two motives. On the one hand, the model under analysis is quite simple; on the other hand, not many mutations have been introduced in the model (about 60). For these reasons, when either applying this technique to more complex models, or simply inserting more mutations, an increment in the temporal cost (both in simulation and analysis times) could be expected.

E. Comparison of the Analysis Results

Tables VII–X contain the analysis results calculated in the four injection campaigns.

It is difficult to compare the outcomes obtained with the three injection techniques, because both their nature and the fault models that can be injected with each technique are very different. However, it is possible to compare the results got with the old and new proposals of both saboteurs and mutants injection techniques.

In relation to saboteurs, we can conclude that the increment in the number of fault models injectable with the new approach has provoked a raise in the number (and thereby, also in the percentage) of activated faults. These new faults affect the system making the FTMs to work harder, as the higher values in the detection and recovery coverages reflect. This allows a better checking of the FTMs.

The trend in new mutants is, to a certain extent, similar to saboteurs. Although the percentage of activated errors is lower, the propagated errors are more harmful than with dynamic mutants (notice again the increase of detection and recovery coverages). Obviously, the 100% of activated errors in dynamic mutants is not realistic at all. So, any incorrect operation of the synchronization mechanism used there must have occurred. A reason for the increment in the detection and recovery coverages can be that the automation of new mutants has allowed injecting faults in more targets than with the old approach.

VII. Conclusion

In this paper, new methods to implement and use saboteurs and mutants into VHDL models in an automatic way have been proposed.

The new models of saboteurs fix some problems of ambiguity that the previous approach had. These problems prevented their automatic insertion. Moreover, the new models have been implemented in such a way that they diminish the overhead, by reducing the number of signals required to manage bidirectional saboteurs. Another enhancement respect to prior models is that they allow to inject more fault models. The numeric results of comparing both proposals do not reflect these improvements. Instead, a slight temporal overhead has been introduced. Anyway, the overall temporal cost (the sum of simulation and analysis times) of this technique is affordable with modern computers.

The advantages of the new proposal to implement mutants are especially relevant: it is easy to automate and reduces notably the spatial overhead. But its main success is to shrink considerably the temporal overhead. In the experiments carried out, the overall temporal cost is equivalent to the obtained with the new saboteurs.

References

[1] C. Constantinescu, "Impact of deep submicron technology on dependability of VLSI circuits," in *Proc. DSN*, 2002, pp. 205–209.

[2] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on soft error rate of combinational logic," in *Proc. DSN*, 2002, pp. 389–398.

[3] C. Constantinescu, "Neutron SER characterization of microprocessors," in *Proc. DSN*, 2005, pp. 754–759.

[4] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Trans. Softw. Eng.*, vol. 16, no. 2, pp. 166–182, Feb. 1990.

[5] *Fault Injection Techniques and Tools for VLSI Reliability Evaluation*, A. Benso and P. Prinetto, Eds. Norwell, MA: Kluwer Academic, 2003.

[6] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: The MEFISTO tool," in *Proc. FTCS*, 1994, pp. 356–363.

[7] V. Sieh, O. Tschäche, and F. Balbach, "VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions," in *Proc. FTCS*, 1997, pp. 32–36.

[8] D. Gil, J. Gracia, J. C. Baraza, and P. J. Gil, "A study of the effects of transient fault injection into the VHDL model of a fault-tolerant microcomputer system," in *Proc. IOLTW*, 2000, pp. 73–79.

[9] D. Gil, J. Gracia, J. C. Baraza, and P. J. Gil, "Impact of faults in combinational logic of commercial microcontrollers," in *Lecture Notes in Computer Science*. Heidelberg, Germany: Springer-Verlag, 2005, pp. 379–390.

[10] *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076–1993, 1994.

[11] A. M. Amendola, A. Benso, F. Corno, L. Impagliazzo, P. Marmo, P. Prinetto, M. Rebaudengo, and M. S. Reorda, "Fault behavior observation of a microprocessor system through a VHDL simulation-based fault injection experiment," in *Proc. EURO-DAC/EURO-VHDL*, 1996, pp. 536–541.

[12] J. Boué, P. Pétillon, and Y. Crouzet, "MEFISTO-L: A VHDL-based fault injection tool for the experimental assessment of fault tolerance," in *Proc. FTCS*, 1998, pp. 168–173.

[13] S. Ghosh and T. J. Chakraborty, "On behavior fault modeling for digital design," *J. Electron. Test.*, vol. 2, no. 2, pp. 135–151, Jun. 1991.

[14] J. R. Armstrong, F.-S. Lam, and P. C. Ward, "Test generation and fault simulation for behavioural models," in *Performance and Fault Modelling with VHDL*, J. M. Schoen, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1992, pp. 240–303.

[15] T. A. DeLong, B. W. Johnson, and J. A. Profeta, III, "A fault injection technique for VHDL behavioral-level models," *IEEE Des. Test Comput.*, vol. 13, no. 4, pp. 24–33, Dec. 1996.

[16] W. Mao and R. K. Gulati, "Improving gate level fault coverage by RTL fault grading," in *Proc. ITC*, 1996, pp. 150–159.

[17] P. Sanchez and I. Hidalgo, "System level fault simulation," in *Proc. ITC*, 1996, pp. 732–740.

[18] P. A. Thaker, V. D. Agrawal, and M. E. Zaghloul, "A test evaluation technique for VHDL circuits using register-transfer level fault modeling," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 22, no. 8, pp. 1104–1113, Aug. 2003.

[19] R. Leveugle, "A new approach for early dependability evaluation based on formal property checking and controlled mutations," in *Proc. DATE*, 2005, pp. 260–265.

[20] H. R. Zarandi and S. G. Miremadi, "Dependability evaluation of altera FPGA-based embedded systems subjected to SEUs," *Microelectron. Reliab.*, vol. 47, no. 2–3, pp. 461–470, Feb.-Mar. 2007.

[21] G. C. Cardarilli, F. Kaddour, A. Leandri, M. Ottavi, S. Pontarelli, and R. Velazco, "Bit flip injection in processor-based architectures: A case study," in *Proc. IOLTW*, 2002, pp. 117–127.

[22] J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil, "A prototype of a VHDL-based fault injection tool: Description and application," *J. Syst. Arch.*, vol. 47, no. 10, pp. 847–867, Apr. 2002.

[23] D. Gil, J. C. Baraza, J. Gracia, and P. J. Gil, "VHDL simulation-based fault injection techniques," in *Fault Injection Techniques and Tools for VLSI Reliability Evaluation*, A. Benso and P. Prinetto, Eds. Dordrecht, The Netherlands: Kluwer Academic, 2003, ch. 4.1, pp. 159–176.

[24] D. Gil, J. Gracia, J. C. Baraza, and P. J. Gil, "Study, comparison and application of different VHDL-based fault injection techniques for the experimental validation of a fault-tolerant system," *Microelectron. J.*, vol. 34, no. 1, pp. 41–51, Jan. 2003.

[25] T. Riesgo and J. Uceda, "A fault model for VHDL descriptions at the register transfer level," in *Proc. EURO-DAC/EURO-VHDL*, 1996, pp. 462–5467.

[26] J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil, "Improvement of fault injection techniques based on VHDL code modification," in *Proc. HLDVT*, 2005, pp. 19–26.

Juan-Carlos Baraza received the M.S. and Ph.D. degrees in computer engineering from the Universidad Politécnica de Valencia, Valencia, Spain, in 1993 and 2003, respectively.

He joined the Departamento de Informática de Sistemas y Computadores (DISCA), Universidad Politécnica de Valencia, in 1994, where he is currently an Associate Professor. He is also a member with the Grupo de Sistemas Tolerantes a Fallos (GSTF), DISCA, since 1994. His research interests include design and implementation of digital systems, design and validation of fault-tolerant systems and fault injection.

Joaquín Gracia received the B.S. degree in computer science and the M.S. and Ph.D. degrees in computer engineering from the Universidad Politécnica de Valencia, Valencia, Spain, in 1995, 1997, and 2004, respectively.

He is currently an Assistant Professor with the Departamento de Informática de Sistemas y Computadores (DISCA), Universidad Politécnica de Valencia. He is member with the Grupo de Sistemas Tolerantes a Fallos (GSTF). His research interests include design and implementation of digital systems, design and validation of fault-tolerant systems, and VHDL-based fault injection.

Sara Blanc received the B.S. degree in computer science and the M.S. and Ph.D. degrees in computer engineering from the Universidad Politécnica de Valencia, Valencia, Spain, in 1995, 1998, and 2004, respectively.

She is currently an Assistant Professor with the Departamento de Informática de Sistemas y Computadores (DISCA), Universidad Politécnica de Valencia. She is member with the Grupo de Sistemas Tolerantes a Fallos (GSTF). Her research interests include design and implementation of digital distributed systems, design, and validation of real-time fault-tolerant systems and fault injection.

Daniel Gil received the B.S. degree in electrical and electronic physics from the Universitat de València, Valencia, Spain, in 1985, and the Ph.D. degree in computer engineering from the Universidad Politécnica de Valencia, Valencia, Spain, in 1999.

He is an Associate Professor with the Departamento de Informática de Sistemas y Computadores (DISCA), Universidad Politécnica de Valencia. His research interests include design and validation of fault-tolerant systems, fault injection, dependability in VLSI, and reliability physics.

Pedro-J. Gil (M'92) received the B.S. degree in electrical and electronic engineering and the Ph.D. degree in computer engineering from the Universidad Politécnica de Valencia, Valencia, Spain, in 1985 and 1992, respectively.

He is a Professor and head of the Departamento de Informática de Sistemas y Computadores (DISCA), Universidad Politécnica de Valencia. His research interests include the design and implementation of real-time fault-tolerant systems, the validation of fault-tolerant systems by fault injection, and the design and implementation of digital systems (including hardware-software codesign). He teaches courses in digital design, computer networks, and fault-tolerant systems. He has authored or coauthored more than 80 research papers in these areas.