Study, comparison and application of different VHDL-based fault injection techniques for the experimental validation of a fault-tolerant system

D. Gil, J. Gracia*, J.C. Baraza, P.J. Gil

Departamento de Informática de Sistemas y Computadores (DISCA), Grupo de Sistemas Tolerantes a Fallos (GSTF), Escuela Universitaria de Informática, Universidad Politécnica de Valencia, Camino de Vera, s/n, Valencia 46022, Spain

Revised 20 June 2002; accepted 25 July 2002

Abstract

In this work different VHDL-based fault injection techniques (simulator commands, saboteurs and mutants) have been compared and applied in the validation of a fault-tolerant system. Some extensions and implementation designs of these techniques have been introduced. As a complement of these injection techniques, a wide set of fault models (including several non-usual models) have been implemented. We have injected both transient and permanent faults on the system model, using two different workloads, with the help of a fault injection tool that we have developed. We have studied the pathology of the propagated errors, measured their latencies, and calculated both detection and recovery coverages. Results show that coverages for transient faults can be obtained quite accurately with any of the three techniques. This enables the use of different abstraction level models for the same system. We have also verified significant differences in implementation and simulation cost between the studied injection techniques.

© 2002 Elsevier Science Ltd. All rights reserved.

Keywords: Dependability; Experimental validation; Fault injection techniques; VHDL simulation; Fault-tolerance mechanisms

1. Introduction

Fault injection is a validation technique of fault tolerant systems (FTSs) which is being increasingly consolidated and applied in a wide range of fields, and several automatic tools have been designed [1,2]. Fault injection is defined in the following way [3].

Fault injection is the validation technique of the Dependability of Fault Tolerant Systems, which consists in the accomplishment of controlled experiments where the observation of the system’s behaviour in presence of faults is induced explicitly by the written introduction (injection) of faults in the system.

Fault injection techniques in the hardware of a system can be classified in three main categories:

- **Hardware implemented fault injection.** It is accomplished at physical level, disturbing the hardware with parameters of the environment (heavy ion radiation, electromagnetic interferences, etc.) or modifying the value of the integrated circuit pins.

- **Software implemented fault injection.** The objective of this technique consists in reproducing at software level the errors that would have been produced upon occurring faults in either the hardware or the software. It is based on different practical types of injection, such as the modification of memory data, or the mutation of the application software or the lowest service layers (at operating system level, for example).

- **Simulated fault injection.** In this technique, the system under test is simulated in another computer system. Faults are induced altering the logical values of the model elements during the simulation.

This work is framed in the simulated fault injection, and particularly in the simulation of models in VHDL. Fig. 1 shows a classification of different techniques that can be used to implement simulated fault injection into VHDL models [4].
Simulator commands technique is based on the use of the simulator commands to modify the value of the model signals and variables without altering the VHDL code [5]. VHDL code modification techniques change the model, adding saboteurs [5–7] or using mutants of the model components [5,8,9].

Other techniques are implemented extending the VHDL language, either by adding new data types and signals, or modifying the VHDL resolution functions [10,11]. The new data types and signals defined include the fault behaviour description. Nevertheless, these techniques require the introduction of ad hoc compilers and control algorithms to manage the language extensions.

The objective of the present work is to use different VHDL-based fault injection techniques in the validation of a fault-tolerant system, showing the advantages and drawbacks of each one. To do this, we have introduced some extensions and implementation designs of these techniques. We have also implemented a wide set of fault models, including some non-usual models. This paper completes previous work on this subject [12–14].

The distribution of this paper is as follows. In Section 2 we describe the injection techniques used, remarking our own contributions. Section 3 shows and justifies the implemented fault models. In Section 4 we summarise the features of our fault injection tool. In Section 5 we present some experiments performed, indicating the computer system model used and the parameters of the injection experiments. In Section 6 we show some significant results obtained. Finally, in Section 7 we explain some general conclusions and future work.

2. VHDL-based fault injection techniques

In this section, we show the different VHDL-based fault injection techniques used in this work (simulator commands, saboteurs and mutants). In every case, we will describe briefly the technique and present some extensions and new designs.

2.1. Simulator commands

This technique is based on the use of the simulator commands to modify the value of the model signals and variables [5].

The way that faults are injected depends on the injection place. To inject faults on signals, the following sequence of pseudo-commands must be performed:

1. Simulate_Until [injection instant]
2. Modify_Signal [name] [faulty value]
3. Simulate_For [fault duration]
4. Restore_Signal [name]
5. Simulate_For [observation time]

This sequence is thought to inject transient faults, which are the most common and difficult to detect [15]. To inject permanent faults, the sequence is the same, but omitting steps 3 and 4. To inject intermittent faults, the sequence consists in repeating steps 1–5, with random separation intervals.

The sequence of pseudo-commands to inject faults on variables is:

1. Simulate_Until [injection instant]
2. Assign_Variable [variable name] [fault value]
3. Simulate_For [observation time]

The operation is similar to the injection on signals, but in this case there is no control of the fault duration. This implies that it is not possible to inject permanent faults on variables using simulator commands.

The sequence of commands needed to carry out the injection (for both transient and permanent faults) can be included in a macro file, where the elements between brackets will be passed to the macro as parameters. This means that the injection conditions can be varied without modifying the command code.

It is interesting to point out that, from the point of view of the injection procedure, VHDL generics are managed as ‘special’ variables. This enables the injection of some non-usual fault types, such as delay faults [16] (by modifying timing generics). See Section 3 for more details about the fault models applied with this technique.

2.2. Saboteurs

A saboteur is a special VHDL component added to the original model [5]. The mission of this component is to alter the value, or timing characteristics, of one or more signals when a fault is injected, remaining inactive during the normal operation of the system.

In Ref. [17] saboteurs are classified into: serial simple, serial complex and parallel. We have extended these types (by designing bi-directional saboteurs), and adapted some models to buses [13,14]. Considering this new classification, we have built a saboteur library, including the following types of saboteurs (Fig. 2):

(a) Serial simple saboteur. It interrupts the connection between an output (driver) and its corresponding receptor (input), modifying the reception value.

(b) Serial simple bi-directional saboteur. It has two input/output signals, plus a read/write input that determines the perturbation direction.
Serial complex saboteur. It interrupts the connection between two outputs and their corresponding receptors, modifying the reception values.

Serial complex bi-directional saboteur. It has four input/output signals, plus a read/write input that determines the perturbation direction.

n-Bit unidirectional simple saboteur. It is used in unidirectional buses of n bits (address and control). It is composed of n serial simple saboteurs.

n-Bit bi-directional simple saboteur. It is used in bi-directional buses of n bits (data and control). It is composed of n bi-directional serial simple saboteurs.

n-Bit unidirectional complex saboteur. It is used in unidirectional buses of n bits (address and control). It is composed of n/2 serial complex saboteurs.

n-Bit bi-directional complex saboteur. It is used in bi-directional buses of n bits (data and control). It is composed of n/2 bi-directional complex saboteurs.

Faults can be injected on the signals which connect components in structural models. The internal architecture of the saboteurs can be behavioural or structural. The behavioural design is basically a process whose sensitivity list contains the control and input/output signals. The structural design is based on the use of multiplexers [4]. Next, we will summarise the design of one of the saboteurs mentioned above.

2.2.1. An example: design of a serial simple bi-directional saboteur

As shown in Fig. 2(b) besides signals I and O, this saboteur has a Control signal to activate the injection, and whose activation determines both the injection instant and fault duration. R/W signal determines the data transfer direction. The selection of the fault model can be made using external Selection signals. Control, R/W and Selection signals are managed at run time by simulator commands. Fig. 3 shows a more detailed scheme of the saboteur and the timing of Control signal.

The behavioural architecture for the saboteur basically consists in a process activated by I, O, R/W and Control signals. If Control is activated, the fault is injected as selected. The perturbation can affect to O or I, depending on the value of R/W. If Control is not activated, O(I) is updated with the new value of I(O), according to the normal operation of the system (without faults). A simplified scheme of this architecture, written in VHDL pseudo-code is:

architecture behaviour of serial_simple_bi-directional_saboteur
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 <= 1;
else
I <= O;
end if;
end if;
end process;
end architecture;
The function fault_type_selection can be easily implemented with a case structure asking for the value of the Selection signal. To ensure single faults, only the rising edge of Control signal must be considered. Obviously, signals I and O must be declared as inout ports in the saboteur entity to be bi-directional. Table 1 shows the possible expressions for $f_{inj}(input, selection)$ that we have used, according to the fault type chosen.

Although the architecture shown is behavioural, it is also possible to implement a structural architecture. It can be made using two serial simple saboteurs in opposition, controlled by the R/W signal. Fig. 4 shows this design.

The design of the serial complex saboteur is merely a generalisation of previous architecture, considering two extra fault models which are becoming increasingly important in CMOS VLSI ICs: short and bridging. For more details about the design of bi-directional saboteurs, saboteurs in buses, and parallel saboteurs, see Ref. [4].

2.3. Mutants

A mutant is a component that replaces another component. While inactive, it works like the original component, but when activated it behaves like the component in presence of faults [5].

We have generated mutants modifying the VHDL code in behavioural descriptions. We have considered the eight fault models exposed in Ref. [9] (summarised in Table 2), as well as other fault models that include modifications of synchronisation and timing clauses (after and wait clauses) [8].

A good solution to implement mutants technique is to use the VHDL configuration mechanism [5]. Nevertheless, in this case the architecture-to-component binding is static, that is, after compiling a specific configuration it remains unaltered during the simulation [18]. For this reason, only permanent faults can be injected in this way. But the injection of transient mutants is a subject of great interest, as it could supply transient fault models with a big semantic power.

The implementation of transient faults by means of mutants technique requires to carry out dynamic instantia-

### Table 1
Fault models implemented in the serial simple bi-directional saboteur

<table>
<thead>
<tr>
<th>Fault model</th>
<th>$f_{inj}(i, o, \text{selection})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuck-at 0</td>
<td>‘0’</td>
</tr>
<tr>
<td>Stuck-at 1</td>
<td>‘1’</td>
</tr>
<tr>
<td>Bit-flip, pulse$^b$</td>
<td>not(I)</td>
</tr>
<tr>
<td>Open-line$^c$</td>
<td>‘Z’ (high impedance)</td>
</tr>
<tr>
<td>Delay</td>
<td>I after delay, delay &gt; 0</td>
</tr>
<tr>
<td>Indetermination</td>
<td>‘X’</td>
</tr>
<tr>
<td>Stuck-open$^a$</td>
<td>‘0’ after $t_{retention}$</td>
</tr>
</tbody>
</table>

$^a$ I or O, depending of R/W value.

$^b$ In combinational logic.

$^c$ Exclusively permanent faults.

### Table 2
Fault models in mutants by modifying syntactical units in behavioural descriptions

<table>
<thead>
<tr>
<th>Fault name</th>
<th>Code modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuck-then</td>
<td>Replacement of the condition by true</td>
</tr>
<tr>
<td>Stuck-else</td>
<td>Replacement of the condition by false</td>
</tr>
<tr>
<td>Assignment control</td>
<td>Disturbing an assignment operation</td>
</tr>
<tr>
<td>Dead process</td>
<td>Elimination of the sensitivity list of a process</td>
</tr>
<tr>
<td>Dead clause</td>
<td>Elimination of a clause in a case</td>
</tr>
<tr>
<td>Micro-operation</td>
<td>Disturbing an operator</td>
</tr>
<tr>
<td>Local stuck-data</td>
<td>Disturbing the value of a variable, constant or signal in an expression</td>
</tr>
<tr>
<td>Global stuck-data</td>
<td>Elimination of all value modifications of a variable or signal in an architecture</td>
</tr>
</tbody>
</table>

Fig. 3. Serial simple bi-directional saboteur and fault activation.

Fig. 4. Structural design of a serial simple bi-directional saboteur.

Fig. 5. Transient mutant implementation. Dynamic modification of the architecture using guarded assignments in blocks.
In Ref. [19], a method to accomplish this objective by means of guarded blocks is suggested. A guarded assignment is an assignment expression on a signal, conditioned by a boolean expression (guard expression or guard predicate) [20]. If the guard expression is true, the assignment is executed. If not, a null assignment is generated. Next we introduce a particular implementation of this idea [13,14].

Fig. 5 shows a scheme of this method to execute the dynamic switching between the fault-free architecture and a mutant architecture of the system. If \( \text{elec} = 1 \), block 1 is enabled. This block is associated to configuration 1 (fault-free), that assigns the fault-free architecture to the global system entity. If \( \text{elec} = 2 \), block 2 is enabled. This block is associated to configuration 2 (mutated), that assigns the mutant architecture to the global system entity. Dynamic block activation involves also multiplexing the system structural architecture signals. \( S_1 \) or \( S_2 \) are assigned to \( S \) according to the guarded signal.

Fig. 6 shows the fault-free and mutant architectures. It has been considered the general case of a structural architecture with components that can be structural or behavioural. A number \( N_i \) of different mutated architectures can be assigned to the same component \( i \), but only one is selected at a time. The binding component-mutant architecture is specified in the declaration of the mutated architecture. Mutants \( (i, 1), (i, 2), \ldots, (i, N_i) \) have been obtained by modifying the syntactical structures of the behavioural descriptions. If there are various processes, one of them is randomly selected. Individual faults have been considered: in the mutant configuration, only a mutant of one component is specified, and the mutant includes a single fault model.

The sequence of pseudo-commands that implement the dynamic activation of architectures shown in Fig. 5 is:

1. Guarded signal \( \leq 1 \)
2. Simulate_Until \( [\text{injection instant}] \)
3. Save_State
4. Write_State in mutated architecture
5. Guarded signal \( \leq 2 \)
6. Simulate_For \( [\text{fault duration}] \)
7. Save_State
8. Write_State in fault-free architecture
9. Simulate_Until \( [\text{simulation time}] \)
10. Write in trace file

The pseudo-commands are based on some simulator commands. Thus, \( \text{Save_State} \) consists in saving the signals and variables of the architecture in a file; \( \text{Write_State} \) consists in modifying the values of the signals and variables of the architecture using simulator commands (\( \text{Modify_Signal}, \text{Assign_Variable} \)).

To inject permanent faults, steps 6–8 must be omitted. The sequence of simulator commands can be included in an injection macro file, where the elements between brackets will be passed to the macro as parameters. Fig. 7 explains the timing diagram of the sequence of pseudo-commands during the simulation. \( T_{\text{inj}} \) is the injection instant, \( \Delta T_{\text{inj}} \) is the fault duration, \( T_{\text{simul}} \) is the overall simulation time and \( \text{elec} \) is the guarded signal.

3. Fault models

We have aimed to include in our injection tool (Section 4) a variety of fault models that would represent real physical faults that occur in integrated circuits (ICs).
Although the most used models are stuck-at (0, 1) (for permanent faults) and bit-flip (for transient faults), as VLSI ICs integration density rises, it becomes more necessary to introduce newer, more complex models. Table 3 summarises the fault models that we have planned to apply in every injection technique.

The models have been deduced from the physical causes and mechanisms implied in the occurrence of faults, at physical and electronic level [4,21,22]. Next, we briefly describe the physical causes and mechanisms implied in these fault models, showing their relationship with the common stuck-at and bit-flip.

Permanent faults are related to irreversible physical defects in the circuit, so they remain indefinitely. These defects can be originated during the manufacturing process (lattice, mask-layout, package, etc.) or the normal operation. In this case, a number of wear-out mechanisms (electromigration, thin-oxide breakdown, hot-electron trapping, electrical stress, etc.) can occur over the long term, revealed initially as intermittent faults until they finally provoke a permanent fault. On the other hand, manufacturing faults can also produce subsequent faults during the circuit operation.

Transient faults are temporal or non-permanent faults that can appear during the operation of a circuit due to different causes, either internal or external. In contrast to permanent faults, transient faults do not introduce a physical defect in the circuit. The treatment of these faults is difficult (which serves to emphasise their importance) because they cannot be located spatially, and their duration is short. Moreover, they do not have a well-defined model, due to the variety of local and external phenomena that can originate them.

Pulse (in combinational logic), bit-flip (in storage) and indetermination models allow to represent transient physical faults of different types: transient in power supply, cross-talk, electromagnetic interferences (light, radio, etc.), temperature variation, α-radiation and cosmic radiation (the last two mechanisms are very important in space and avionics applications).

The delay model allows to represent physical faults due to transients in power supply, which can alter the switching delay of MOS transistors, or the charge/discharge delay of parasitic capacitances in input/output connections. Also, due to the high integration densities and work frequencies, the skin and Miller effects [22] provoke alterations in the RC time constant, originating timing violations.

About the fault models used in mutants technique, we have applied changes in syntactical elements of the language, at algorithmic level. As commented in Section 2.3, we have considered the models proposed in Refs. [8,9]. However, fault modelling for algorithmic behavioural descriptions is an open research field [23,24], and new fault models must be studied and tested in the future.

### 4. The fault injection tool

We have developed a VHDL-based injection tool (called VFIT [25]) to validate fault-tolerant systems. It is a simple and easily portable tool, suitable for injection experiments in models at gate, register and chip level. It runs on an IBM-PC (or compatible) platform. The tool is composed by a series of elements designed around a commercial VHDL simulator [26]. With this tool, we intend to be able to inject faults using three important VHDL-based injection techniques: simulator commands, saboteurs and mutants.

About the fault timing, both transient, permanent and intermittent faults can be injected, being possible to choose among different probability distribution functions (Uniform, Exponential, Weibull and Gaussian) to determine both the injection instant and duration. The tool can inject a wide range of fault models, surpassing the classical models of stuck-at and bit-flip.

### 5. Fault injection campaign

In this section we will show a set of experiments carried out to validate a FTS using the three fault injection techniques explained above.

We have built the VHDL model of a fault-tolerant microcomputer. The system is duplex with cold stand-by sparing, parity detection and watchdog timer [27]. Each component is modelled by a behavioural architecture, usually with one or more concurrent processes. Both main and spare processors are an enhanced version of the MARK2 processor [28].

The parameters of the injection experiments are summarised [14]:

1. **Injection technique.** Simulator commands, saboteurs and mutants.
2. **Number of faults.** \( n = 3000 \) single faults per experiment. With this value, the confidence intervals (calculated with a 95% confidence level) for latency and coverage estimators have acceptable values.
3. **Fault types.** Transient and permanent faults.

<table>
<thead>
<tr>
<th>Injection technique</th>
<th>Transient</th>
<th>Permanent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Simulator commands</strong></td>
<td>Bit-flip, pulse,(^a)</td>
<td>Stuck-at (0, 1), indetermination, open-line, delay</td>
</tr>
<tr>
<td></td>
<td>indetermination, delay</td>
<td>indetermination, open-line, delay</td>
</tr>
<tr>
<td><strong>Saboteurs</strong></td>
<td>Bit-flip, pulse,(^a)</td>
<td>Stuck-at (0, 1), indetermination, open-line, delay, short, bridging, stuck-open</td>
</tr>
<tr>
<td></td>
<td>indetermination, delay</td>
<td></td>
</tr>
<tr>
<td><strong>Mutants</strong></td>
<td>Syntactical changes</td>
<td>Syntactical changes</td>
</tr>
</tbody>
</table>

\(^a\) In combinational logic.
4. **Injection instant.** Defined according to a Uniform distribution function in the range \([0, t_{\text{workload}}]\), where \(t_{\text{workload}}\) is the execution time of the workload.

5. **Fault duration** (for transient faults). Defined according to a Uniform distribution function in the range \([0.1T_{\text{cycle}}, 10.0T_{\text{cycle}}]\), where \(T_{\text{cycle}}\) is the CPU clock cycle.

6. **Workload.** Common workloads in fault injection experiments are well-known numeric and sorting algorithms. In this case, the workloads implemented are the calculus of the arithmetic series of \(k\) integer numbers, and bubblesort algorithm.

7. **Fault location.** For simulator commands technique, any atomic signal and variable of the model. For saboteurs technique, the signals of the top structural architecture. For mutants technique, syntactical elements in the behavioural architecture of the components. No faults are injected in the spare CPU, since it is off while the system is working properly.

8. **Fault models.** We have applied the models specified in Table 3.

9. **Simulation duration.** It includes the execution time of the workload and the recovery time with the spare CPU \((t_{\text{simul}} = t_{\text{workload}} + t_{\text{spare}})\).

10. **Result analysis.** For every injection experiment, two files are compared: the simulation traces with and without faults. The estimated coverages and latencies of the detection and recovery mechanisms are automatically recorded. Fig. 8 shows the fault-tolerance mechanisms predicate graph [29]. It represents the pathology of faults, that is, their evolution since faults are injected until they are detected and eventually recovered.

From the sample data, the following measures are calculated:

- **Percentage of activated faults**
  \[
  P_A = \frac{N_{\text{activated}}}{N_{\text{injected}}} = \frac{N_{\text{activated}}}{n}
  \]
  where \(N_{\text{activated}}\) is the number of activated errors. We define an activated error as an error originated by an activated fault (a fault that produces a change in any model signal or variable), which is propagated to the propagation signals. In our case, the propagation signals are those of the external structural architecture of the system.

- **Error detection coverage**
  \[
  C_d(\text{mechanisms}) = \frac{N_{\text{detected}}}{N_{\text{activated}}}
  \]
  where \(N_{\text{detected}}\) is the number of errors detected by detection mechanisms.

- **Recovery coverage**
  \[
  C_r(\text{mechanisms}) = \frac{N_{\text{detected}, \text{recovered}}}{N_{\text{activated}}}
  \]
  where \(N_{\text{detected}, \text{recovered}}\) is the number of errors detected and recovered by detection and recovery mechanisms.

- **Propagation, detection and recovery latencies.**
  - Propagation latency, \(L_p = t_p - t_{\text{inj}}\), where \(t_p\) is the instant when the error reaches the propagation signals (i.e. the instant when the error is activated), and \(t_{\text{inj}}\) is the injection instant.
  - Detection latency, \(L_d = t_d - t_p\), where \(t_d\) is the instant when the activated error is detected by detection mechanisms.
  - Recovery latency, \(L_r = t_r - t_d\), where \(t_r\) is the instant when the detected error is recovered by recovery mechanisms.

The experiments have been carried out on a PC-compatible with a Pentium® II processor at 350 MHz, and 192 Mb RAM.

6. Results

Figs. 9–14 show the percentage of activated errors, coverages and average latencies in function of the fault duration (transient/permanent faults), the workload and the fault injection technique [12–14].

6.1. Influence of the injection technique

In Fig. 9 we can observe that \(P_A\) raises notably when using saboteurs and mutants. This means that faults injected...
with these techniques have a higher influence in the system operation than those injected with simulator commands technique. This can be due to the fact that saboteurs and mutants techniques have a larger fault model capability, including models that cannot be injected using simulator commands technique (Table 3).

From Fig. 10, no significant variation of $C_d$ is observed. The most important reduction of $C_d$ is observed in permanent mutants.

Fig. 11 does not reflect notable differences in $C_r$ either, except in permanent saboteurs. In this case, $C_r$ falls greatly respect to the other techniques.

With relation to the propagation latency, Fig. 12 shows that $L_p$(commands) > $L_p$(mutants), $L_p$(saboteurs). This is the general trend, except in a few cases, like permanent saboteurs with the bubblesort workload. In general, this indicates that faults injected with saboteurs and mutants techniques take less time to be propagated as activated errors. This can be related to the fault location. In saboteurs technique, faults are injected in signals of the top structural architecture. As propagation signals are also located in the top structural architecture, the propagation latency is low. In the case of mutants, the reason is not so evident. Nevertheless, for transient mutants the reduction of the propagation latency is less pronounced.

Figs. 13 and 14 show $L_d$ and $L_r$. For transient faults, $L_d$(commands) > $L_d$(mutants) > $L_d$(saboteurs). For permanent faults, no clear dependency on the injection technique is observed.

Figs. 12–14 corroborate the results obtained in Ref. [27], showing that $L_p < L_d << L_r$. Check-pointing recovery mechanism contributes greatly to the total latency.

The differences observed in $P_A$, coverages and latencies are due to the changes introduced in the fault models (Table 3) and fault locations (Section 5), that depend on the injection technique.

### 6.2. Influence of the fault duration

In general, the results corroborate the previous results obtained in Refs. [16,27,30]. As the fault duration increases:

- $P_A$ increases (except in mutants). Permanent faults have a higher influence in the system operation.
- $C_d$ rises (except in mutants). Permanent faults are easier to detect than transient faults.
- $C_r$ decreases. A portion of permanent faults affect the spare system performance (even though the faults are not originated in the spare CPU) and it precludes the system recovery.
Lp, Ld and Lr do not reflect a clear dependency on the fault duration. Only Ld clearly rises when increasing the fault duration.

6.3. Influence of the workload

In this case, some relevant differences have been observed:

- \( P_A \) does not reflect important differences, except in mutants.
- \( C_p \) shows significant differences in permanent faults.
- \( C_r \) exhibits significant differences in both transient and permanent faults.
- \( L_p \) does not reflect important differences, except in permanent saboteurs.
- \( L_d \) does not reflect important differences, except in transient mutants.
- \( L_r \) presents enough differences.

It is evident that workload affects notably on Dependability measures, because it constitutes a mechanism that sensitises faults produced in the hardware. It is necessary a more rigorous study of the influence of the workload on the FTSs performance.

6.4. Fault and error pathology

Lastly, Figs. 15–17 show some illustrative examples of the detailed fault and error pathology for the three fault injection techniques, concerning transient faults and using as workload the arithmetic series.

It can be observed that there are a non-negligible number of faults unrecovered or even undetected. Despite the system under test is a fault-tolerant microcomputer, it must be noted that it is an academic model, where the microcomputer has only one internal detection mechanism: parity. This affects the detection coverage values. As recovery begins after detection, recovery coverage is also affected. To improve the coverages, it would be necessary to extend the internal detection mechanisms of the microprocessor, including, for instance, exceptions (Incorrect opcode address, Unused memory, Invalid read address, Invalid write address, Division check, etc.).

6.5. Implementation cost

Simulator commands technique is the easiest technique to implement because the manipulation of signals and variables is performed directly, and it does not require the modification of the VHDL code.

The main drawback of the saboteurs technique is that a number of control signals have to be added to the model. These signals are used to activate one among the set of inserted saboteurs and to indicate the type of perturbation to be injected. This adds an additional complexity to both the model and technique. Although saboteur technique is more difficult to implement than simulator commands, it has a larger fault model capability, as seen in Section 3.

The main problem of transient mutants technique is the high temporal cost of simulations, mainly due to Save_State and Write_State pseudo-commands. Nevertheless, this technique has a big fault-modelling capability because they can use all the syntactic and semantic elements of the language.

Table 4 shows the time needed to make an experiment with 3000 transient faults. The time for simulation and analysis phases is separated. We can see that mutants technique is the most costly, especially in simulation phase.
For transient faults, the values of detection and recovery coverages exhibit little differences. Coverages can be obtained quite accurately with any of the three techniques. This enables to work with models at different abstraction levels for the same system.

- For permanent faults, mutants technique shows some discrepancy respect to the other techniques in the values of detection and recovery coverages. Perhaps it would be necessary a deeper study of the representativeness of fault models for permanent mutants.
- The average values of detection and recovery latencies in saboteurs and mutants techniques are a bit optimistic (lower values) compared to simulator commands technique. The cause can be related to the type of locations perturbed by every technique.
- With relation to implementation cost, simulator commands technique is the easiest to implement. Saboteurs and mutants have a higher elaboration and application cost because of recompilation and simulation times. On the other side, saboteurs technique enlarges the size of trace files due to the increase of control signals in the model. Mutants technique increases the simulation time due to the necessity of saving and restoring the system state when switching architectures.

### 7. Summary: conclusions and future work

In this work, three VHDL-based fault injection techniques have been compared: simulator commands, saboteurs and mutants. Some extensions and implementation designs of these techniques have been exposed, mainly related with bi-directional saboteurs, saboteurs in buses, and transient mutants.

Also, a wide set fault models for all the techniques have been implemented, surpassing the most commonly applied in the literature.

As an application, a fault-tolerant microcomputer system has been validated. It is a structural chip-level model, where the components in the second level are behavioural. Faults have been injected using a fault injection tool that we have developed. We have injected both transient and permanent faults on the system model, using two different workloads. We have studied the pathology of the propagated errors, measured their latencies, and calculated both detection and recovery coverages.

From the experiments and the results achieved, we can obtain some general conclusions. We have verified the good performance of the VHDL simulation-based fault injection technique. It is worth to mention the high controllability and observability of injection experiments. Also, we have shown the usefulness of the injection tool, developed to work on PC platforms and to inject faults at chip-level into VHDL models. About the influence of the different VHDL-based fault injection techniques on Dependability measures, it is interesting to stand out some preliminary results (in this specific model):

- For transient faults, the values of detection and recovery coverages exhibit little differences. Coverages can be

<table>
<thead>
<tr>
<th>Fault injection technique</th>
<th>Simulation phase</th>
<th>Analysis phase (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Series</td>
<td>Bubblesort</td>
</tr>
<tr>
<td>Simulator commands</td>
<td>2 h</td>
<td>3 h</td>
</tr>
<tr>
<td>Saboteurs</td>
<td>2.5 h</td>
<td>6 h</td>
</tr>
<tr>
<td>Mutants</td>
<td>10 days</td>
<td>14 days</td>
</tr>
</tbody>
</table>

This is due to the save and write state processes, needed to execute the dynamic switching between the fault-free architecture and the mutated architecture. About the size of simulation files, files produced when using saboteurs technique are approximately six times larger than the ones obtained with simulator commands and mutants techniques (for both workloads). This is due to the increase of control signals introduced in the model.

### Table 4

Temporal cost for 3000 transient faults (experiments performed on a PC-compatible with a PII processor at 350 MHz and 192 Mb RAM)

About the relationship between the type of system model and the injection techniques, some considerations can be made. If logic or RTL levels are used, the most suitable techniques are simulator commands and saboteurs. Although saboteurs technique is more difficult to implement, it has a larger fault model capability. At algorithmic level (in behavioural models, normal situation in the first stages of the design), both simulator commands and mutants techniques can be used. Although mutants technique is more difficult to implement, it has a larger fault modelling capability, because they can use all the syntactical and semantical capabilities of the VHDL language.

In the future, it is intended to extend these experiments to validate more complex FTSs. In addition, we will focus this research on some injection tool enhancements, such as improving the automation of saboteurs and mutants techniques and the reduction of simulation time in mutants technique. Also, it is important to study in depth the representativeness of fault models, especially in mutants technique. Lastly, another open problem to study is the influence of the workload on Dependability measures, being necessary to consider more benchmark workloads of different types.

### Acknowledgements

This work has been partially funded by the European research project DBENCH IST-2000-25425.
References