A Prototype of a VHDL-Based Fault Injection Tool

J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil
Grupo de Sistemas Tolerantes a Fallos (GSTF)
Departamento de Informática de Sistemas y Computadores (DISCA)
Universidad Politécnica de Valencia. Spain
e-mail: { jcbaraza, jgracia, dgil, pgil}@disca.upv.es

Abstract

This paper presents the prototype of an automatic and model-independent fault injection tool, to use on an IBM-PC (or compatible) platform. The tool has been built around a commercial VHDL simulator. With this tool, both transient and permanent faults, of a wide range of types, can be injected into medium-complexity models. Another remarkable aspect of the tool is the fact that it is able to analyse the results obtained from the injection campaigns, in order to study the Error Syndrome of the system model and/or validate its Fault-Tolerance Mechanisms. Some results of a fault injection campaign carried out to validate the Dependability of a fault tolerant microcomputer system are shown. We have analysed the pathology of the propagated errors, measured their latencies, and calculated both error detection and recovery latencies and coverages.

1. Introduction

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

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 writing introduction (injection) of faults in the system.

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

- *Physical fault injection*: It is accomplished at physical level, disturbing the hardware with parameters of the environment (heavy ions radiation, electromagnetic interferences, etc.) or modifying the value of the pins of the integrated circuits.
- *Software Implemented Fault Injection (SWIFI)*: The objective of this technique, also called Fault Emulation, consists of reproducing at software level the errors that would have been produced upon occurring faults in the hardware. It is based on different practical types of injection, such as the modification of the memory data, or the mutation of either the application software or the lowest service layers (at operative system level, for example).
- *Simulated fault injection*: In this technique, the system under test is simulated in other computer system. The faults are induced altering the logical values during the simulation.

This paper describes a tool for injecting faults in VHDL simulation models. Previous works in this area can be divided in two groups of techniques [3], as it can be seen in Figure 1.

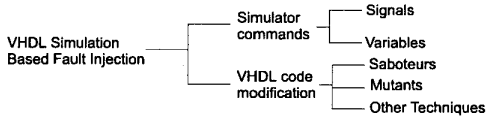

Figure 1: VHDL-based fault injection techniques.

0-7695-0719-0/00 $10.00 © 2000 IEEE

Simulator commands technique is based on the use of the simulator commands in order to modify the value of the model signals and variables [4]. VHDL code modification techniques change the model, adding *saboteurs* [5] or using *mutants* of the model components [6].

MEFISTO (Multi-level Error/Fault Injection Simulation TOol) [7] settled up the basis of the theory of the VHDL-based fault injection, although it only injected faults using the simulator commands technique. Two improved versions of MEFISTO were implemented by the LAAS (MEFISTO-L [8]) and the Chalmers University of Technology (MEFISTO-C [9]). These tools can inject faults adding *saboteurs* to the model, or manipulating the signals and/or variables of the model. They use the Vantage Optium VHDL Simulator, being done this simulation in parallel, in a network of UNIX workstations. Besides the classical fault models (stuck-at, bit-flip), they implement new types of faults: open-line (high impedance), indetermination, etc.

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

Since the good perspectives of the VHDL-based fault injection to carry out an early validation of the system during the design phase, the GSTF decided to build an injection tool with these features:

- Utilisable on an IBM-PC (or compatible) platform.
- General, which could be used on any medium-complexity model.
- Automatic, in order to get that all injection phases were performed by a unique application.
- Complete, which would:
  a) Implement the main injection techniques: simulator commands, *saboteurs* and *mutants*.
  b) Allow to inject as much fault models as possible.
- Universal, easy to use.

The tool presented in this paper can inject faults into VHDL models (at gate, register and chip level), and analyse their effects. The analysis can be the study of the error syndrome of the model or its validation.

The distribution of this paper is as follows. In section 2 we describe the injection tool, explaining summarily its block diagram. In section 3, 4 and 5 we present some experiments performed with the tool, showing respectively the computer system model used, the parameters of the injection campaign, and the results obtained. Finally, in section 6 we explain some general conclusions and future enhancements of the tool.

2. The fault injection tool

The injection tool presented in this paper is based on a previous prototype showed in [12]. That prototype was made in VHDL, and it was composed by individual utilities that were executed using a commercial VHDL simulator running on a PC (or compatible) platform: V-System, by Model Technology [13].

The fault injection consisted of three independent phases:

- **Experiment Set-up**. Here, with the help of a basic graphic interface, the injection parameters were specified and written in an ASCII file.
- **Simulation**. In this phase, two operations were carried out. First, a set of macros was automatically written: one performed an error-free simulation of the model; the others had the commands needed to perform the injection campaign. Then, the macros were executed by the VHDL simulator, obtaining a set of simulation traces: one with an error-free behaviour (with no faults injected), and *n* with a fault injection performed, being *n* the number of faults injected.
- **Readouts**. The error-free simulation trace was compared to the *n* fault-injected simulation traces, analysing their differences and extracting the Dependability parameters of the system model (mainly latencies and coverages).

This version had two drawbacks:

1. The tool was not actually a compact program, but a set of independent utilities simulated into a VHDL simulator.

2. Some of the parameters and function procedures were model dependent, and had to be changed manually when the model changed or a different analysis was performed.

However, the tool was helpful to make some studies about the validation of the fault tolerance by means of fault injection on VHDL models [14] [15]. For this reason, we decided to improve it, in order to get a new version, more automated, and model independent.

2.1. Features

With this tool, we intend to be able to inject faults using three important VHDL simulation techniques: simulator command based, *saboteurs* and *mutants*.

At this point, the injection using simulator commands is fully developed, the *saboteurs* technique is being developed, and the *mutants* technique is under study [16].

About the fault timing, both transient and permanent 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.

With respect to the fault models used, they depend on the injection technique and the abstraction level of the system model. We have worked mainly with VHDL models at gate, register and chip level.

Thus, in the simulator commands technique, the fault models are:

- Transient fault models: Stuck-at (0, 1), indetermination, bit-flip, and delay.
- Permanent fault models: Stuck-at (0, 1), indetermination, delay, and open-line (high-impedance).

In the *saboteurs* technique, the fault models are the same as for the simulator commands, plus short, bridging, and stuck-open, which are permanent fault models.

2.2. Block diagram

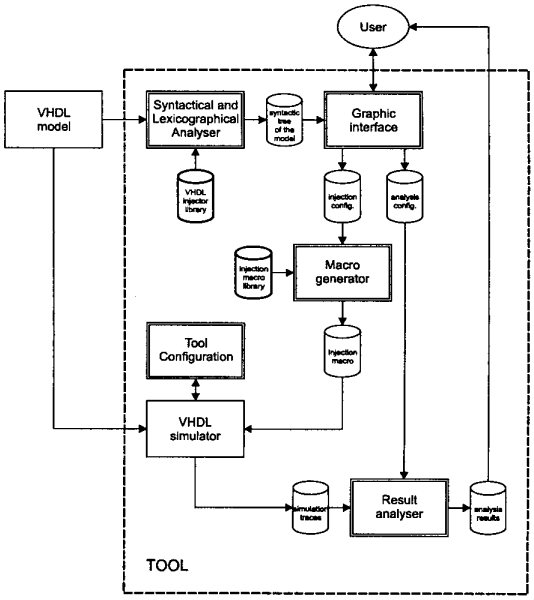

Figure 2: Block diagram of the fault injection tool.

Figure 2 shows the block diagram of the new version of the tool. As it can be seen, it is composed by a series of elements designed around a commercial VHDL simulator. Now, we are using the new simulator by Model Technology, ModelSim [17].

A remarkable aspect of the new version of the tool is the fact that the VHDL simulator has been integrated. This has been possible due to the fact that ModelSim can be executed in batch mode much more easily than V-System.

To reach the automation of the tool, all the operation elements have been implemented in C++, using the Borland C++ Builder 3 compiler. This way, we have got a unique executable file that manages all the injection phases, activating the appropriate element in every step of the operation.

Next, the tool elements are described summarily.

Tool configuration

The mission of this module is to configure the tool, considering that the simulator is a part of it, setting both the tool and simulator parameters. The most important aspect is to get the matching of the libraries used.

Syntactical and Lexicographical Analyser

The mission of this module is to scan all the model files, in order to get its *Syntactic Tree*. Basically, this tree includes all the possible injection targets of the model. The tree structure reflects the hierarchical architecture of the model, including components, blocks and processes.

Depending on the injection technique used, the type of injection targets may vary:

- **Simulator commands**: The atomic signals and variables of the system model, belonging to either structural or behavioural architectures.
- ***Saboteurs***: The atomic signals belonging to a structural description of the system model.
- ***Mutants***: The syntactical elements of the model VHDL code in behavioural architectures.

VHDL injector library

This library holds a set of predefined VHDL *saboteur* components that will be included in the model when this technique is used.

Graphic interface

This utility is a complete set of window-based menus, with which the user can specify all the parameters needed to perform an injection campaign, including both the related to the injection and analysis.

The most important injection parameters are:

- Campaign parameters: injection technique (simulator commands, *saboteurs* or *mutants*), number of injections, injection targets, fault models for every injection target, and the injection instant and duration timing.
- System model dependent parameters: clock cycle and system workload.

Basically, the analysis parameters are:

- Set the objective of the injection campaign: error syndrome or FTSs validation.
- If an error syndrome analysis is carried out: the fault classification and the error classification clauses.
- If a FTS validation is performed: the error detection and recovery clauses.

Obviously, many of these parameters are model dependent, mainly the ones related to the injection analysis. For this reason, the *Syntactical and Lexicographical Analyser* becomes a very important part of the tool.

Injection macro library

This library is composed by a set of predefined macros, used to inject on the model signals and variables according to the injection technique used.

Up to date, there are macros to inject using the simulator commands and *saboteur* techniques.

Macro generator

This module is actually an injection manager, because it controls the injection process. Using the *injection configuration* generated by the *graphic interface*, it

1. generates a series of *injection macros*, in order to perform an error-free simulation and the number of fault-injected simulations specified in the parameters, and
2. invokes the simulator to make it run the macros, obtaining the *simulation traces*.

VHDL simulator

As indicated before, in this version of the tool we have used ModelSim, the most recent VHDL simulator for IBM PC (or compatible) by Model Technology.

This is a simple and easy-to-use event-driven simulator. When activated, the simulator executes a file with macros and generates the output trace of the simulation.

This simulator has some advantages over its predecessor, V-System:

1. Due to its new modular structure, it can be executed in batch mode from the tool kernel more easily.
2. Its enhanced macro language allows implementing automatically new fault types, particularly bit-flip and delay.

Result analyser

This module takes as input the *analysis configuration* generated by the *graphic interface*. According to those parameters, it compares the error-free simulation trace to all the fault-injected simulation ones, looking for any mismatches.

Depending on the analysis type, the objective of the comparison is different:

- In case of an error syndrome analysis, it increments an internal counter associated to the injected fault and measures the propagation latency.

  At the end of the analysis, the main results obtained are the error classification and the propagation latencies for every fault type specified in the fault classification.
- In case of a FTS validation, if no mismatch has been found, it considers that the injected fault has produced a non-effective error. If a mismatch was found, it searches for the assessment of the detection clauses to determine whether the error has been detected or not. If not, the injected fault has produced a non-detected error. In case the error has been detected, it searches for the assessment of the recovery clauses to determine whether the error has been recovered or not.

At the end of the analysis, the Fault-Tolerance Mechanisms predicate graph [18] (see Figure 3) is fulfilled. From the graph, the detection and recovery coverages and latencies can be calculated.

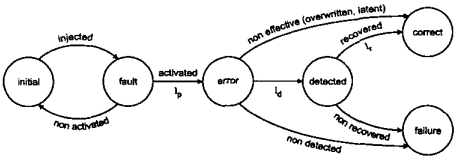

Figure 3: Fault-Tolerance Mechanisms predicate graph.

3. System model

We have built the VHDL model of a fault-tolerant microcomputer, whose block diagram is shown in Figure 4. The system is duplex with cold stand-by sparing, parity detection and watchdog timer.

The structural architecture of the model is composed by the following components:

- Main and spare CPUs (CPUA and CPUB, respectively).
- RAM memory (MEM).
- Output parallel port (PORTOUT).

- Interrupt controller (SYSINT).
- Clock generator (CLK).
- Watchdog timer (WD).
- Pulse generator (GENINT).
- Two back-off cycle generators (TRGENA, TRGENB).
- Two AND gates (PAND2A, PAND2B).

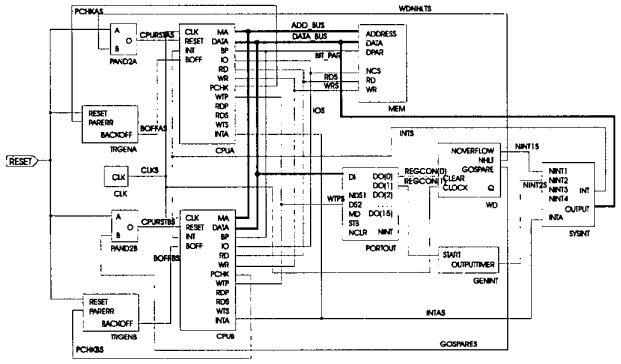

Figure 4: Block diagram of the computer system.

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 [19].

The description of this system is around 1500 lines of VHDL code. The code is divided into 10 entities, 11 architectures and 1 package, excluding the STD and IEEE libraries. In addition, 416 bytes are used to store the machine code executed by the CPU.

As mentioned before, several fault-tolerance mechanisms have been added to increase the dependability of the system. The error detection mechanisms include the parity check and program control flow check by a watchdog timer. The error recovery mechanisms include the introduction of a back-off cycle when parity error detection, checkpointing when errors are detected by the watchdog timer, and starting the spare processor in case of permanent errors. A more detailed description of the detection and recovery mechanisms can be seen in [15].

4. Fault injection experiments

The aim of these experiments was to study the response of the system in presence of transient and permanent faults. The injection conditions of the campaign are similar of those used in [20], with some differences in the fault types and duration. Summarily, these are:

1. **Number of faults**: n = 3000 faults per injection experiment.
2. **Workload**: Calculus of the arithmetic series of n integer numbers, with n=6.
3. **Fault types**: The injected faults are transient, stuck-at 0, stuck-at 1, bit-flip, indetermination or delay, and they may affect the signals and the variables in the model. There have been also injected permanent faults of type stuck-at 0, stuck-at 1, indetermination, delay and open-line (high impedance).
4. **Injection place**: Any atomic signal and variable of the model. Faults are not injected in the spare CPU, since it is off while the system is working properly.
5. **Injection instant**: Uniform [0, $t_{workload}$], where $t_{workload}$ is the execution time of the workload.
6. **Simulation duration**: The simulation duration includes the execution time of the workload and the recovery time with the spare CPU ($t_{Simul} = t_{Workload} + t_{Spare}$).

7. **Fault duration**: It has been intended to inject "short" faults, with a duration equal to a fraction of the clock cycle (the most common ones, as described in [21]), as well as longer faults, which will ensure in excess the propagation of the errors to the detection signals. Three cases have been considered:
   a) transient faults with a duration generated randomly in the range [0.01T-1.0T], where T is the CPU clock cycle,
   b) transient faults with a fixed duration equal to 100T, and
   c) permanent faults.
8. **Analysis results**: From the sample data, the following parameters are obtained:
   - Percentage of activated faults, P_A.
   - Error detection coverage. We have distinguished two types of coverage estimators:
     * Coverage of the detection mechanisms

$$C_{d(mechanisms)} = \frac{N_{Detected}}{N_{Activated}}$$

     * Global system coverage

$$C_{d(system)} = \frac{N_{Detected} + N_{Non-effective}}{N_{Activated}}$$

   - Recovery coverage. Divided also in two types of coverage estimators:
     * Coverage of the recovery mechanisms

$$C_{r(mechanisms)} = \frac{N_{Detected\_recovered}}{N_{Activated}}$$

     * Global system coverage

$$C_{r(system)} = \frac{N_{Detected\_recovered} + N_{Non-effective}}{N_{Activated}}$$

   - Propagation, detection and recovery latencies.

5. Results

Table 1: Percentage of activated errors, coverages and latencies related to the fault duration.

	Duration		
Parameters	**[0.1T-10.0T]**	**100T**	**permanents**
P_A (%)	23.47	33.40	39.77
$C_{d\,(mec)}$ (%)	29.40	43.41	47.02
$C_{d\,(sys)}$ (%)	96.31	91.52	88.52
$C_{r\,(mec)}$ (%)	24.29	35.13	18.36
$C_{r\,(sys)}$ (%)	91.19	83.23	59.85
L_p (ns)	979	8811	7770
L_d (ns)	31527	45261	40781
L_r (ns)	109915	101879	121464

Table 1 shows the percentage of activated errors, the coverages and the average latencies in function of the fault duration. It can be observed that, as fault duration decreases:

- P_A decreases. Short duration faults have lesser influence in the system operation.
- $C_{d\,(mechanisms)}$ decreases. Short duration faults are more difficult to detect and recover. However, $C_{d\,(sys)}$ follows the opposite behaviour. This is due to the raise in the percentage of non-effective errors.
- C_r follows the same behaviour as C_d for durations = Uniform [0.01T-1.0T] and 100T, with slightly smaller percentages. This means that the recovery mechanisms are working well, and almost any detected error is recovered. But C_r decreases notably for permanent faults. The rea-

son can be that a portion of the permanent faults affects the spare system performance (even though the faults are not originated in such a CPU) and it precludes the system recovery.

- $L_p < L_d << L_r$. The values of average latencies don't seem to have a clear dependency on the fault duration.

In short, these results reproduce the the influence of the fault duration showed in [15].

Tables 2 and 3 show the detached contribution of the different detection and recovery mechanisms to coverage and latency results.

Table 2: Percentage of detected errors and average detection latency related to the detection mechanisms.

Detection Mechanism	% Detected Errors			Average Latency (ns)		
	[0.1T-10.0T]	100T	Perm.	[0.1T-10.0T]	100T	Perm.
Parity	61.35	38.85	42.42	3021	6062	6210
WDT	38.65	61.15	57.58	76779	70165	66255

Table 2 shows that:

- For fault durations in range [0.1T-10T], % Parity > % WDT. The most effective detection mechanism is Parity. For duration = 100T and permanent faults, it is WDT.
- L_d (Parity) << L_d (WDT). Parity presents a quite lower latency than WDT for all fault durations.

Table 3: Percentage of recovered errors and average recovery latency related to the recovery mechanisms.

Recovery Mechanism	% Recovered Errors			Average Latency (ns)		
	[0.1T-10.0T]	100T	Perm.	[0.1T-10.0T]	100T	Perm.
Back-off	22.81	4.26	6.39	13916	81333	54810
Checkpoint	5.85	44.32	1.83	35932	46577	25194
Spare	71.34	51.42	91.78	146667	151244	128022

Table 3 shows that:

- For fault durations in range [0.1T-10.0T], % Spare > % Back-off >> Checkpoint.
- For fault duration equal to 100T, % Spare > % Checkpoint >> Back-off.
- Permanent faults provoke a higher percentage of permanent errors, which active the spare CPU.

In range [0.1T-10.0T], L_r (Back-off) < L_r (Checkpoint) << L_r (Spare). For 100T and permanent faults, L_r (Checkpoint) < L_r (Back-off) < L_r (Spare).

6. Summary. Conclusions and future work

In this work, a VHDL-based fault injection tool to run on PC platforms is shown, describing its main components and overall performance. The tool is easy to use, versatile, and appropriate to medium-complexity system models.

We have verified the usefulness of the tool carrying out an injection campaign into a VHDL model of a 16-bit fault-tolerant microcomputer system running a workload. We have injected transient and permanent faults (in groups of 3000 in each experiment) on the model signals and variables, using the simulator commands technique. The objectives have been to study the pathology of the propagated errors, measure their latency, and calculate the detection and recovery coverages.

The results obtained in the injetion campaign can be used to improve the design of detection and recovery mechanisms to optimise the values of coverage and latency.

It is intended to complete this work in a short term in the following aspects:

- Injection tool enhancements:
  - Implementation of automatic fault injection using *saboteurs* and *mutants*.
  - Improvement of the user interface, offering the results in standard formats, such as tables or graphics.
  - Reduction of the simulation time, by starting it just before the occurrence of the fault.

  * Migration of the tool to more powerful VHDL simulators.
- Testing new injection techniques which can modify the VHDL code, starting from the idea of *saboteurs* (at structural level) and *mutants* (at behavioural level).
- Applying the fault injection tool to the validation of more complex and real FTSs.

References

[1] M. Sueh, T. Tsai, and R.K. Iyer, "Fault Injection Techniques and Tools", IEEE Computer, April 1997, pp. 75-82.

[2] J. Arlat, "Validation de la Sûreté de Fonctionnement par Injection de Fautes. Méthode-Mise en Oeuvre-Application", Thèse présentée à L'Institut National Polytechnique de Toulouse, Rapport de Recherche LAAS N° 90-399, Décembre 1990.

[3] D. Gil, "Validación de Sistemas Tolerantes a Fallos me-diante inyección de fallos en modelos VHDL", Tesis Doctoral, DISCA-UPV, 1999.

[4] E. Jenn, "Sur la validation des systèmes tolérant les fautes: injection de fautes dans des modèles de simulation VHDL", Thèse, Rapport LAAS N° 94-361, 1994.

[5] J. Boué, P. Pétillon, Y. Crouzet, and J. Arlat, "Early Experimental Validation of Fault Tolerance: the VHDL-based Fault Injection tool MEFISTO-L", DeVa ESPRIT Project 20072, 2nd year Report, 1997.

[6] J.R. Armstrong, F.-S. Lam, and P.C. Ward, "Test generation and Fault Simulation for Behavioural Models", Performance and Fault Modelling with VHDL (J.M. Schoen ed.), Englewood Cliffs, Prentice Hall, 1992, pp. 240-303.

[7] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool", Proc. 24th Int. Symp. on Fault-Tolerant Computing (FTCS-24), Austin, Texas (USA), June 1994, pp. 66-75.

[8] J. Boué, P. Pétillon, and Y. Crouzet, "MEFISTO-L: A VHDL-Based Fault Injection Tool for the Experimental Assessment of Fault Tolerance", Proc. 28th Int. Symp. on Fault-Tolerant Computing (FTCS-28), Munich (Germany), June 1998, pp. 168-173.

[9] M. Rimén, J. Ohlsson, and S. Svensson, "MEFISTO: Multilevel Error and Fault Injection Simulation Tool. User's Manual", Chalmers University of Technology, Gothenburg (Sweden), 1997.

[10] T.A. DeLong, B.W. Johnson, and J.A. Profeta III, "A fault Injection Technique for VHDL Behavioral-Level models", IEEE Design and Test of Computers, Vol. 13, N° 4, Winter 1996, pp. 24-33.

[11] V. Sieh, O. Tschäche, and F. Balbach, "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions", Proc. 27th Int. Symp. on Fault-Tolerant Computing (FTCS-27), Seattle, Washington (USA), June 1997, pp. 32-36.

[12] D. Gil, J.V. Busquets, J.C. Baraza and P.J. Gil, "A Fault Injection Tool for VHDL Models", Fastabs of the 28th Int. Symp. on Fault-Tolerant Computing (FTCS-28), Munich (Germany), June 1998, pp. 72-73.

[13] Model Technology, "V-System/Windows PC User´s Manual. Version 4.6", 1997.

[14] D. Gil, J.C. Baraza, J.V. Busquets, and P.J. Gil, "Fault Injection into VHDL Models: Analysis of the Error Syndrome of a Microcomputer System", Proc. 24th Euromicro Conference (EUROMICRO 98), Vol. 1, Västerås (Sweden), August 1998, pp. 418-424.

[15] D. Gil, R. Martínez, J.C. Baraza, J.V. Busquets, and P.J. Gil, "Fault Injection into VHDL Models: Experimental Validation of a Fault Tolerant Microcomputer System", Proc. 3rd European Dependable Computing Conference (EDCC-3), Prague (Czech Republic), September 1999, pp.191-208.

[16] J. Gracia, D. Gil, J.C. Baraza, and P.J. Gil, "Application of Different VHDL-Based Fault Injection Techniques to the Validation of a Fault-Tolerant Microcomputer System", Workshops and Abstracts of the International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8), New York (USA), June 2000, pp. B–54-B–55.

[17] Model Technology, "ModelSim EE/PLUS Reference Manual", 1998.

[18] J. Arlat, A. Costes, Y. Crouzet, J.C. Laprie, and D. Powell, "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems", IEEE Transactions on Computers, Vol. 42, N° 8, August 1993, pp. 913-923.

[19] J.R. Armstrong, "Chip-Level Modelling with VHDL", Prentice Hall, 1989.

[20] 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", Proc. 6th IEEE Int. On-Line Testing Workshop (IOLTW 2000), Palma de Mallorca (Spain), July 2000, pp. 73-79.

[21] H. Cha, E.M. Rudnick, G.S. Choi, J.H. Patel, and R.K. Iyer, "A fast and accurate gate-level transient fault simulation environment", Proc. 23rd Int. Symp. on Fault-Tolerant Computing (FTCS-23), Toulouse (France), June 1993, pp. 310-319.