A Functional Verification based Fault Injection Environment

A Functional Verification based Fault Injection Environment

A. Benso*, A. Bosio*, S. Di Carlo*, R. Mariani+
*Politecnico di Torino ,Dipartimento di Automatica e Informatica, Torino, Italy
+Yogitech S.p.A, San Martino Ulmiano Pisa (Italy)
{alfredo.benso, alberto.bosio, stefano.dicarlo}@polito.it
riccardo.mariani@yogitech.com

Abstract

Fault injection is needed for different purposes such as analyzing the reaction of a system in a faulty environment or validating fault-detection and/or fault-correction techniques. In this paper we propose a simulation-based fault injection tool able to work at different abstraction levels and with user-defined fault models. By exploiting the facilities provided by a functional verification environment it allows to speed up the entire fault injection process: from the creation of the workload to the analysis of the results of injection campaigns. Moreover, the adoption of techniques to optimize the fault list significantly reduces the simulation time. Being the tool targeted to the validation of dependable systems, it includes a way to extract information from the Failure Mode and Effect Analysis and to correlate fault injection results with estimates.

1. Introduction

Dependability analysis [1] is a concern for Integrated Circuits (IC) designers and manufacturers since erroneous behaviors were first reported in space applications in the mid 70's. Phenomena such as alpha particles or heavy ion strikes may lead to dramatic consequences and their occurrence increases with technology downscaling [2]. It is thus mandatory to early analyze the behavior of digital circuits employed in critical applications (avionics, automotive, etc.) affected by these phenomena [3-5]. Fault-injection experiments have demonstrated as one of the most effective approaches for IC dependability evaluation [5-9]. The international norm IEC61508 [10], regulating the requirements of safety-related systems, highly recommends fault-injection in all steps of the development process. Nevertheless, setting up a fault-injection environment is not trivial and requires to tune different parameters (e.g. the fault model, the fault list, the workload, the outputs used as readout points, and the way experimental results are interpreted) that can strongly influence the coherency and the meaningfulness of the final results.

Different fault-injection techniques have been proposed and used in the past. They can be grouped in three different categories: (i) simulation-based, (ii) software-based, and (iii) hardware-based. Simulation-based fault injection [11-13], injects faults in a simulative model of the target system. It allows early and detailed dependability analysis and it can be applied when a prototype is not yet available. Moreover, it actually allows modeling any type of fault. However, it is very time-consuming and the effectiveness depends on the accuracy of the system model. Software-based fault-injection [6-7] [14] targets microprocessor-based systems and resorts to modifications of the software, executed by the microprocessor, to inject faults and to observe their effect. It significantly speeds up the fault injection process. Finally, hardware based fault-injection uses hardware platforms (mainly FPGA based) to inject faults [15-17]. It notably speeds up the injection process w.r.t. simulation and software based approaches; nevertheless, it requires a synthesizable model of the system and sometime it is difficult to apply.

1550-5774/07 $25.00 © 2007 IEEE
DOI 10.1109/DFT.2007.31

This paper exploits the use of coverage-driven functional verification to build an efficient simulation-based fault-injection environment. Coupling fault-injection with functional verification allows to overcome some of the drawbacks affecting existing environments, such as hard reusability or adaptability to different designs and/or fault models. Verification components available on the market can be easily reused as workloads to inject faults, obtaining at the same time design validation and reliability evaluation. Moreover, the use of standard verification languages enables an easy and configurable way to model injected faults and, by adopting dedicated techniques to collapse faults in the fault list, to reduce the fault list size and therefore the required injection time. Finally, coverage-driven functional verification allows to uniquely correlating workloads, operational profiles, fault list, and final measures.

The proposed solution is intended to work with any functional verification EDA tool available on the market. In this paper, we will refer to Specman Elite ® by Cadence™ [18] (Specman for short in the remaining of the paper), together with the IEEE *e* Standard Verification Language [19] as a reference example. This research work has been performed in the framework of an implementation of a complete flow (analysis and validation with fault-injection) to extend Failure Modes and Effects Analysis (FMEA) to System On Chip. This flow has then been used during the design of "robust" microcontrollers for automotive applications [20][21].

The paper is organized as follow: Section 2 introduces the overall architecture of the fault injector environment, Section 3 to 7 detail each required step to setting up the fault-injection flow. Section 8 gives some experimental results, while Section 9 concludes the paper.

2. The fault-injector architecture

This paragraph overviews the overall architecture (see Figure 1) of the proposed fault-injection environment. The main idea is to provide high flexibility and to allow reliability assessment at different stages of the design process and design levels.

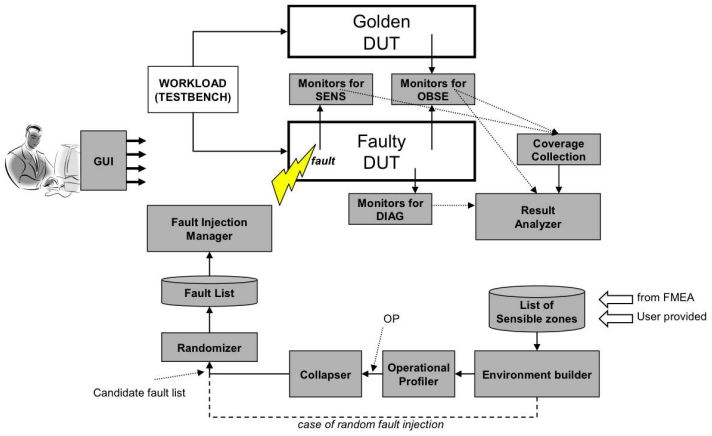


Figure 1: Fault-injection environment overall architecture

The fault injector is basically composed of a fault free (Golden) and a faulty Device Under Test (DUT) simulated in parallel and sharing the same workload. They can be described using any type of hardware description language (e.g. VHDL, Verilog, etc.) provided with a simulator able to interface with the selected functional verification tool. The fault-injection flow starts from a list of sensible zones (SENS), i.e., sites of the DUT where to inject faults, and a set of observation/diagnostic points (OBSE/DIAG), i.e., sites where to observe the effects of the injected faults. They can be either internal nets or external pins of the DUT. An OBSE is a site where to measure the result of an

injection in terms of the difference w.r.t. the corresponding point in the golden DUT. A DIAG is a site where to measure the result of an injection based on the occurrence (or not) of a given event. Typically, DIAGs are outputs of logic blocks inserted to increase the fault tolerance of the system. By monitoring these sites, it is possible to understand their detection/correction capability. This initial input (i.e., list of SENSs, OBSEs, and DIAGs) can be provided either by the user or obtained from a Failure Modes and Effects Analysis (FMEA) [21][22], if available. The FMEA is a methodology to analyze potential dependability problems early in the development cycle when it is easy to find solutions. It provides a list of potential failing points, failure modes and classification of hazards easy to translate into SENSs, OBSEs, and DIAGSs. The Environment builder (see Figure 1) collects this information and setup the initial injection environment needed to generate the target fault list. The proposed architecture allows two different solutions for the generation of the fault list:

- *Random generation*: faults are randomly (through the Randomizer block) selected from the complete list of fault locations;
- *Operational profile based*: faults are identified based on Operational Profiles and then collapsed in order to reduce the overall fault list size and to reduce the final experiment duration. The Operational Profiler and the Collapser are in charge of performing this operation (see Sections 5.1 and 5.2 for the details).

The Fault Injection Manager (see Section 6) finally performs the injection experiments. It runs an iterative process selecting a single fault at a time from the fault list and injecting the fault into the faulty DUT during the application of the target workload. The simulation of the Faulty and the Golden DUTs are then continuously monitored by a set of so called monitors instantiated by the Environmental Builder. The outputs of the monitors are finally used by the Result Analyzer to estimate the final coverage (see Section 7). The remaining sections of the paper will detail the characteristics of the main elements composing the proposed fault-injection environment.

3. Fault model

The choice of the target fault model is a key point in setting up a fault-injection campaign. Real faults are influenced by different factors such as the target system technology and the environmental working conditions and they can be classified in many different ways. The flexibility of the IEEE *e* standard Verification Language [19] adopted by Specman (or any other verification language), can be exploited to model faults. It allows the user to describe complex faulty behaviors, not only at the gate level, but also at higher abstraction levels (e.g. a glitch in a given data bus signal as a consequence of a given condition on the address bus). This is the first advantage gained from the use of the IEEE *e* standard Verification Language together with Specman to build our fault-injection environment. Each fault is modeled with a function called by the fault-injection manger. The function receives different parameters depending on the selected fault model. Figure 2 shows the *e* code modeling a Single Event Upset (SEU). This simple code waits until the injection event (`inj_event`) becomes true and then flips the state of the target location `inj_port`.

```
<'
        inject ()@clk is {
                wait inj_event;
                inj_port = ! inj_port;
        };
'>
```

Figure 2: *e* code modeling a SEU

4. Workload Generator

After selecting the target fault model, another challenge in performing fault-injection experiments is the identification of a meaningful workload to apply to the target DUT during the injections. It is possible to identify two different types of workloads:

- *Mission Oriented:* the experiments are performed to evaluate the dependability of the DUT when executing its "mission" application. In this case the workload can be either partially or totally fixed (it is the application itself, e.g. a software) and parts of the system may not be considered in the experiments because they are not excited by the application;
- *Device-oriented*: the experiments are performed to evaluate the dependability of the device, regardless its mission application. In this case the workload must be generated so to functionally exercise all the parts of the device.

In case of device oriented fault-injection, again the facilities provided by a functional verification tool (i.e., Specman) and in particular by its test generation engine may help generating high quality test benches. In fact, the goal of the workload generation is to come up with a set of patterns able to activate all (or a subset of) the different parts of the target system in different possible ways. It is actually very similar to the goal pursued in a functional verification flow. For example, in Specman, the test generation engine is based on a random generator that can be driven and constrained in a very flexible way, in order to explore corner-cases and particular critical situations. Moreover it is possible to reuse functional verification test benches written using the IEEE *e* standard Verification Language [19] or even different languages (it can also include software). The completeness of the workload is automatically measured by using coverage monitors on sensible zones and observation points as described in Section 7. A workload is considered complete if all the sensible zones are excited at least once, and all the observation point monitors are triggered at least once.

5. Fault List Generator

As already introduced in Section 2, starting from the list of sensible zones it is possible to generate the target fault list for the injection campaign. The idea is to provide the user with two different approaches.

The first possibility is the random generation performed trough the Randomizer block of Figure 1. In this case, a subset of the complete set F of possible faults is randomly chosen to compose the target fault list. Each selected fault $f \in F$ is identified by its fault location (SENS) and injection time. The use of the random approach allows reducing the fault-injection environment setup time but it may not lead to optimal results. Actually, this approach uniformly distributes faults in different locations/execution times regardless the real importance of the target zone for the behavior of the application. The alternative to the random fault generation is an operational profile based fault list generation [23]. An Operational Profile (OP) is a collection of information about relevant fault-free system activities. Traced information is read/write activities associated with signals, or system elements (register, buses, memory elements, etc.), but they may also include other more high level information like the most probable expected sets of inputs that the system or application should receive. Essentially, the purpose of the operational profile is to better understand the conditions in which the system or the application has to work (the workload), and then to analyze this information to target only faults that actually may lead to errors. This approach allows to compact the fault list and to consider non-trivial faults only. In

particular, faults that lead to predictable effects, such as “no effect”, are kept in the fault list (they still contribute to the final measures) but not injected. The achievable fault list reduction factor depends on the workload and on the complexity of the device under test. We can therefore split the fault list generation into three different steps: (i) the OP generation, (ii) the analysis of the OP and the generation of the compacted fault list, (iii) the optional use of the randomizer to randomly select faults from the compacted list. This last step is used only if the size of the compacted fault list is still too high for the computational resources. The following subsections will detail the OP generation and the OP analysis steps.

5.1. Operational Profile Generation

An Operational Profile (OP) is an instrument to optimize the execution time of a fault-injection experiment. It consists of a collection of information (log) about relevant fault-free system activities on each potential fault location of the target system when the selected workload is applied. Logged activities include read/write associated with any element of the DUT model (signals, variables, registers, buses, etc.). Information is collected using simulator breakpoints. The operational profiler generates the script to trace the proper signals (including the breakpoint instructions able to log when a location is accessed by the DUT either for a write or a read operation).

As an example Figure 3 shows part of a VHDL code representing a FIFO core where we want to trace read and write memory accesses, i.e. operations on the `ram` variable. It is easy to see that we have two locations to trace: at row 58 (Figure 3) a write of `ram` occurs whereas at row 66 a read of `ram` occurs. The simulator commands used to log the two locations are shown in Figure 4. This example is based on Cadence NCSIM simulator; it instruments one breakpoint (stop) for each accessed location.

```
56. if (clock'event and clock = '1') then
57.    if ((write_enb = '1') and  (full = '0')) then
58.           ram(w_ptr) := data_in;
59.           empty <= '0';
60.           w_ptr:= (w_ptr + 1) mod (16);
61.           if ( r_ptr = w_ptr ) then
62.                  full <= '1';
63.           end if;
64.    end if;
65.    if ((read_enb = '1') and (empty = '0')) then
66.           data_out <= ram(r_ptr);
67.           full <= '0';
68.           r_ptr := (r_ptr + 1) mod (16);
69.           if ( r_ptr = w_ptr ) then
70.                  empty <= '1';
71.           end if;
72.    end if;
73. end if;
```

Figure 3: VDHL FIFO RAM code

```
set fileid [open "./oplist.txt" w ]
scope -set $HDL_location_path
stop    -line    58    -all    -silent    -continue       -execute    {puts    $fileid
"$HDL_location_path:ram([value %d w_ptr]), write, [time NS]"}
stop    -line    66    -all    -silent    -continue       -execute    {puts    $fileid
"$HDL_location_path:ram([value %d r_ptr]), read, [time NS]"}
```

Figure 4: Simulator script example

The result of this script is reported in Figure 5.

```
:router1:queue_0:fifo_core:ram(0), write, 5350 NS
:router1:queue_0:fifo_core:ram(8), read, 5350 NS
```

Figure 5 : Operational profile

5.2. Operational Profile Generation

The information contained in the OP can be efficiently used to collapse the fault list associated with a given DUT and a given list of sensible zones with a consequent reduction of the simulation time. This is possible by introducing additional constraints to avoid the selection of inactive fault locations.

For example, let us consider transient faults (e.g. SEU). A fault location is sensitive at a given time t_1 if the next operation performed on the same location at time $t_2>t_1$ is a read operation (i.e., it does not overwrite the effect of the fault). Therefore, in case of transient faults, for each fault location, the period between a read and a write operation is inactive (no faults need to be injected). Figure 6 shows an example of an OP analysis. The given OP shows that a selected fault location (e.g. a flip-flop) is read at simulation time 20, 30, and 60 and written at simulation time 10, 40, 50, and 70. Using the constraint previously introduced the only intervals available for the injection are: $10<t_{inj}<20$, $30<t_{inj}<40$ and $50<t_{inj}<60$.

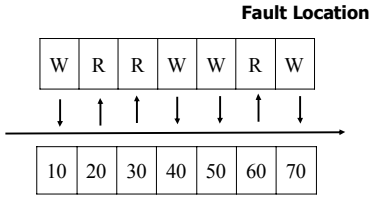


Figure 6: Collapsing example

Injecting in other time instants (e.g. 40 or 50) would be useless since the fault would be overwritten by the next write operation. Moreover, to further reduce the size of the target fault list, the user can specify "condition" signals for injection, called "effect condition" and "no effect condition": a fault must be injected in these zones only if the "effect condition" signal is "true". A detailed explanation of the proposed approach can be found in [24]. Moreover, the user can implement his collapsing rules based on his fault models.

6. Fault-Injection Manager

The Fault-Injection Manager performs the actual injection of faults inside the DUT. It resorts to the full controllability and observability of DUT internal signals provided by the functional verification tool. One fault at a time is injected; based on the fault model the injection engine stops the simulation, injects the fault (i.e., it executes the verification code that model the injected fault as described in Section 3), and then resumes the simulation. At the same time observation points (OBSEs/DIAGs) behavior are logged for later analysis. An interesting improvement is the use of so called "stop run" timers. They control the behavior of each simulation during the injection campaign in order to reduce the total simulation time. Examples of stop timers are: a timer stops the simulation if the simulation time exceeds the expected test bench duration; a timer stops the simulation after a given period after the injection of a given fault if no activity has been detected on the observation/diagnostic points, etc.

7. Result Analyzer

The result analyzer evaluates the result of the fault-injection, i.e., the reaction of the system to the injected faults. The analysis is based on the information collected by the monitors placed on OBSEs and DIAGs (see Section 2). Fault effects can be classified based on the circuit functionality in two ways: "failure" and "no-effect". The failure can

manifest as a "data mismatch" or a "time alteration" between the golden and the faulty DUT. In case of no-effect the error is overwritten or corrected by a fault tolerance mechanism and, it does not propagate in the circuit. In this case, the use of DIAGs (Section 2) allows understanding if the circuit really tolerates the fault. In case of data mismatch the faulty DUT produces a wrong output w.r.t. the golden DUT. In this case the error has been propagated to the output of the circuit generating a wrong behavior. In the last situation, i.e., timing alteration, the circuit produces a correct result but with different timings. Depending on the constraints of the application and the introduced overhead this situation may be acceptable or not.

Another important measure provided by the result analyzer is a so called coverage. In the fault-injection terminology, the coverage is defined as the probability of system recovery when a fault appears, i.e., saying that a system has a fault-coverage of 99% means that over the totality of the injected faults, only 1% resulted in an error or failure. However, given the complexity of modern systems, it is necessary to provide more accurate measures. The traditional coverage is not a real assessment of the system reliability without a correlation with the "accuracy" of the fault list. In general, the smaller the fault-list is, the less accurate the final reliability measures are. We introduce the verification concept of functional coverage defined as a systematic procedure to assess how and how much each verification item, or specification requirement, has been covered by the tests. These verification items are in fact called "coverage items", and they depend on the system architecture and also on the application running on the system itself. We can therefore define a "*cross-coverage*" (cross between functional coverage and fault-injection coverage) as a measure of how many times each coverage item has been hit by a fault, independently if the final effect of such event is a failure or not. If at the end of the process a coverage item has not been cross-covered up to a certain threshold that means the fault list was not enough accurate. A measure of the cross-coverage is also important to identify critical parts of the system for which the OP algorithm was not accurate enough.

8. Experimental Results

To proof the concepts of the proposed tool, we performed a set of experiments on a simple router device injecting SEUs. It accepts data packets on a single input port and, routes the packets to one of three output channels: channel0, channel1, or channel2. Each channel includes a buffer used to store data to send in output. The buffer is implemented as a FIFO 8 x 16 (16 words of 8 bits). As target fault locations we selected the three FIFOs. We performed two sets of experiments both using the operational profile based generation. The first one adopts the collapsing algorithm proposed in Section 5.2, while the second one does no collapse the fault list. Obviously the experiments use the same workload. The aim of the experiments is to show the efficiency of the tool (using the Operational Profiler and the Collapser) in terms of injection time and percentage of fault effects[1]. The selected workload is device oriented and generated using Specman itself. It is very important to underline that the components used to generate the workload for the injection campaigns are exactly the same adopted during the functional verification, allowing high reusability. The resulting workload generates an equal number of packets for each channel of the router. The time overhead introduced by the generation of the operational profile, w.r.t. a fault free

[1] In a campaign without collapsing the fault list; the percentage of no effect faults should be higher than under collapsed fault list

simulation, is equal to 16,3%. The operational profile generation shows that each ram is accessed 3216 times during the simulation.

In order to reduce the simulation time we applied the collapsing algorithm introduced in Section 5.2. Figure 7 sketches the performance of the algorithm in terms of predicted fault effects. The chart shows for each fault location the number of candidate faults and, the relative number of forecasted "no effect" faults. It is easy to see that we have a 50% reduction of the number of real injections. The "unknown effect" corresponds to the actual fault-injection list. We have 1622 injections for each ram (1622 * 3 = 4886) instead of 3216 injections (3216 * 3 = 9648). The results of the injection campaign with the collapsed fault list are in Figure 8.a. The required time to perform the 4886 simulations is about 5 hours. To show the effectiveness of the collapsing procedure, Figure 8.b. shows the injection results using the complete fault list (i.e. 3216 injections for each ram). The required simulation time was in this case of 11 hours (more than twice the simulation time with the collapsed fault list). Moreover, we obtained a higher number of "no effect" w.r.t. a data mismatch violations.

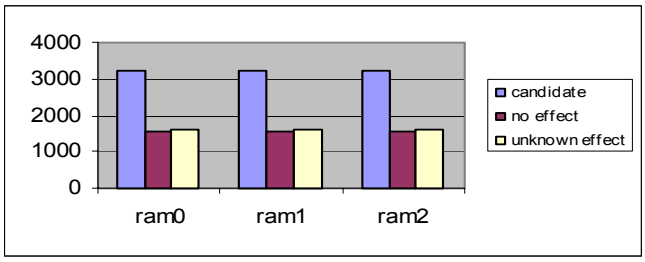
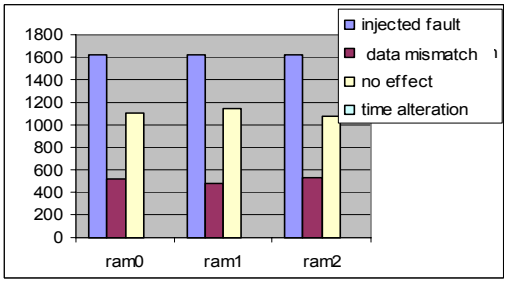
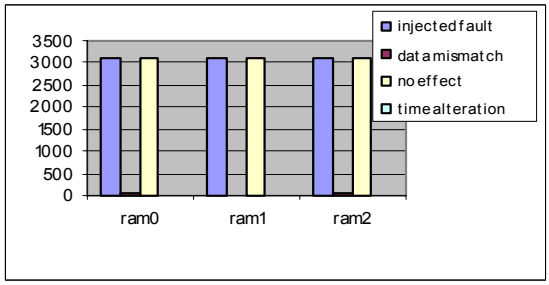


Figure 7: Collapsing results



(a) results with collapsed fault list



(b) results without collapsed fault list

Figure 8: Injection campaign results

More complex experiments with the tool have been performed during the validation of a real safety critical system based on a 32-bit RISC processor. An example of results extrapolated by these fault-injection campaigns are: 125s of CPU time for total operational profile extraction, 464K total lines of operational profile, 6s of CPU time for total collapsing, 25K faults after collapsing, 56.6s of average single injection time (the workload of this example was very complex).

9. Conclusions

This paper presented a fault-injection environment based on the functionalities provided by EDA functional verification tools and languages. The main innovative features of the proposed tool are: the integration of verification and fault-injection methodologies in the same environment; the possibility to work with different description languages and at different abstraction levels; the use of a standard

verification languages to model faults in a systematic and well-defined way; the use of Operational Profiles to generate effective and non trivial fault lists and finally the use of concepts of coverage to deliver precise measures about the fault-injection experiment completeness. Experimental results show the efficiency of the proposed flow when adopting collapsing rules.

10. References

[1] Laprie J.C. “Dependable computing and Fault Tolerance: Concept and Terminology”, IEEE Twenty-Fifth International Symposium on Fault-Tolerant Computing, June 1995, pp. 2.

[2] Vanhauwaert P. et al., “Reduced Instrumentation and Optimized Fault Injection Control for Dependability Analysis”, IEEE International Conference on Very Large Scale Integration, October 2006, Date, pp. 391--396.

[3] Benso A. and Prinetto P “Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation”, Kluwer Academic Publishers, 2003, ISBN 1-4020-7589-8.

[4] Koopman P. et al, “Toward Middleware Fault Injection for Automotive Networks”, Robotic Institute, June 1998.

[5] Arlat J. et al., “Fault Injection and Dependability Evaluation of Fault-Tolerant Systems”, IEEE Transactions on Computers, Vol. 42, No. 8, August 1993, pp. 913--923.

[6] Benso A. et al., “EXFI: A Low-cost Fault Injection System for Embedded Microprocessor-Based Boards” ACM Transaction On Design Automation of Electronic Sytems, Vol. 3, No. 4, 1998, pp. 626—634.

[7] Carreira J. et al., "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers", IEEE Transactions on Software Engineering, Vol. 24 No. 2, pp. 125--136, February 1998.

[8] Mei-Chen H. et al., "Fault Injection Techniques and Tools", IEEE Transaction on Computer, Vol. 30, No. 4, 1997, pp. 75--82.

[9] Benso A. et al., “An integrated HW and SW fault injection environment for real-time systems”, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 1998, pp. 117—122.

[10] CEI International Standard IEC 61508, 1998-2000.

[11] Jenn E. et al., “Fault Injection into VHDL Models: the MEFISTO Tool”, IEEE 24th International Symposium on Fault-Tolerant Computing, June 1994, pp. 66—75.

[12] Cha H. et al., G. S. “A gate level simulation environment for alpha-particle-induced transient faults”, IEEE Transaction on Computers, vol. 45, November 1996, pp. 1248--1256.

[13] Cardarilli G. C. et al., “Bit flip injection in processor based architectures: a case study”, IEEE International On-Line Testing Workshop, 2002, pp. 117—127.

[14] Velazco R. et al., “Predicting Error Rate for Microprocessor-Based Digital Architectures Through C.E.U. (Code Emulating Upsets) Injection”, IEEE Transaction on Nuclear Science, Vol. 46, No. 6, 2000, pp. 2405—2411.

[15] Lopez-Ongil C. et al., “Autonomous transient fault emulation on FPGAs for accelerating fault grading”, 11th IEEE International On-Line Testing Symposium, 2005, pp. 43—48.

[16] Civera P. et al., “FPGA-based Fault Injection for Microprocessor Systems”, IEEE Asian Test Symposium, 01, pp. 304-309.

[17] Lima F. et al., “On the use of VHDL Simulation and Emulation to Derive Error Rates”, 6th European Conference on Radiation and its Effects on Components and Systems, 2001, pp. 253—260.

[18] Specman elite home page. [Online]. Available: http://www.cadence.com/products/functional_ver/specman_elite.html.

[19] IEEE Standard e Functional Verification Language, IEEE Std. 1647, 2006.

[20] Mariani R. et al., “Cost-effective Approach to Error Detection for an Embedded Automotive Platform”, SAE 2006 World Congress & Exhibition, April 2006, Detroit, MI, USA.

[21] Mariani R. et al., “Using An Innovative Soc-Level Fmea Methodology To Design In Compliance With IEC61508”, IEEE Design Automation and Test Conference in Europe ,2007.

[22] McDermott R. E. et al., Basics of FMEA, Quality Resources, 1996.

[23] Musa J.D. “Operational profiles in software-reliability engineer”, IEEE Software, Vol. 10, No. 2, March 1993, pp. 14—32.

[24] Benso A.; et al., “Fault-list collapsing for fault-injection experiments”, IEEE Proc. of Reliability and Maintainability Symposium, 1998, pp. 383 – 388.