SSIRI 2009 Student Doctoral Program
A fast and flexible Platform for Fault Injection and Evaluation in Verilog-based Simulations

David Kammler, Junqing Guan, Gerd Ascheid, Rainer Leupers, Heinrich Meyr
Institute for Integrated Signal Processing Systems, RWTH Aachen University, Germany
Email: kammler@iss.rwth-aachen.de

Abstract

This paper presents a complete framework for Verilog-based fault injection and evaluation. In contrast to existing approaches, the proposed solution is the first one based on the Verilog programming interface (VPI). Due to standardization of the VPI, the framework is—in contrast to simulator command based techniques—independent from the used simulator. Additionally, it does not require recompilation for different fault injection experiments like techniques modifying the Verilog code for fault injection. The feasibility of the VPI-based approach is shown in a case study.

Index Terms

Fault injection, failure evaluation, Verilog, Verilog programming interface (VPI)

1. Introduction

With ever shrinking technology sizes in the semiconductor industry, unwanted parasitic effects resulting from process variations, capacitive coupling, heat flux, reduced supply voltage, energetic radiation and electromagnetic interference etc. are increasing [1]. These can lead to unreliable systems when no protection mechanisms are in place. However, careful fault tolerant design still promises to achieve reliable systems from unreliable technologies.

The development of fault tolerant design calls for evaluation and validation of fault tolerant mechanisms at an early design stage. Therefore, simulation based fault injection (FI) appears to be a good candidate to guide the designer. Widely used are FI techniques aiming on simulations based on hardware description languages (HDLs) [2]–[8] as they offer simulations on register transfer level (RTL) as well as gate level, enabling a certain tradeoff between simulation speed and accuracy.

This work has been supported by the UMIC Research Centre, RWTH Aachen University.

Our literature review shows that existing approaches either need to modify the HDL code or utilize simulator commands for FI. The former require a recompilation, which can result in long durations for statistical analysis running hundreds or thousands of individual simulations. As we will show, the latter are not well suited for Verilog simulations, since faults cannot be injected in all places of the model. Therefore, in this work we introduce a novel Verilog FI technique overcoming these drawbacks. It is based on the standardized Verilog programming interface (VPI). A complete framework for Verilog-based fault injection and evaluation is presented. Due to its modularity it is easily extendable to other HDLs and FI techniques. Moreover, support for new types of faults can simply be added to the framework by providing a C function implementing the fault behavior.

The rest of this paper is structured as follows. Section 2 gives a brief overview of related work in this area. Implemented fault models are outlined in Section 3. Section 4 presents the novel *VPI-based* FI technique. In Section 5 we introduce the failure classification and evaluation used in our platform, which is detailed in Section 6. The usability of our framework is shown with results of a case study in Section 7. The paper is concluded and future work is outlined in Section 8.

2. Related Work

Fault injection techniques can be classified in three main categories, i.e. *physical fault injection*, *software fault injection* and *simulated fault injection* [6], [9].

Initially, most studies related to FI on a prototype of a system were reliant on *physical FI*, which introduces faults directly in the hardware of the target system by disturbing the working environment of hardware (electromagnetic interferences, heavy ion radiation, etc.) or modifying the value of its pins. Several problems, such as controllability and repeatability, are associated with this technique [10]. Besides, the injection devices are expensive and it requires long and costly design cycles because measurements can only be performed on real silicon. MESSALINE [11] and FIST [12] are examples

for tools using physical injection technique.

Software FI is usually achieved by changing the contents of memory and registers to emulate the consequences of hardware faults. It enables control of the injection place. Furthermore, the injection and evaluation can be carried out repeatedly and reproducible. However, there are also several shortcomings, such as limited injection locations and poor time-resolution [13]. Tools making use of software FI are for example FERRARI [14], FTAPE [15] and DOCTOR [16].

In *simulated FI* the whole system behavior is modeled and imitated using simulation. It is particularly attractive as it can provide not only early checks in the design process of fault tolerance mechanisms, but also controllability and accessibility to all the component models. HDLs are widely used in this field as they enable simulations on RTL as well as on gate level. VHDL and Verilog are two prevalent HDLs. Tools using simulated FI targeting VHDL are for instance MEFISTO [2], MEFISTO-L [3], DEPEND [4], VERIFY [5] and VFIT [6]. INJECT [7] is based on Verilog and SINJECT [8] supports both HDLs.

Table 1: Existing simulated fault injection techniques

HDL	Simulator commands		HDL code modification		
VHDL	*Signals*	*Variables*	*Saboteur*	*Mutant*	*Others*
Verilog	-		*Mutant*		

Generally, simulated FI techniques can be classified into two main categories, i.e. the *code-modification* (CM) technique and the *simulator-command* (SC) technique, as shown in Table 1.

Saboteurs and *Mutants* are the two most common CM FI techniques (e.g. [2], [6], [8]). The first one is based on adding components, called *saboteurs*, to the HDL model, while the second replaces the original component with a modified one, called *mutant*. Further HDL CM techniques can be found in [5], [17]. All these methods have a significant limitation which obligates the modification of the source code and therefore requires recompilation. Depending on the number, variety and size of experiments, the time consumed by recompilation can be significant.

The SC FI technique is based on the modification of the signal values through simulator commands. In VHDL different commands need to be used when faults are injected on *signals* or *variables* [6]. Our literature review does not reveal any SC FI technique for Verilog. We will show in the next section that faults cannot be injected in all places of the Verilog model using SCs. One major benefit of simulator commands is that it is not necessary to modify the source code. Therefore, no extra time is consumed by recompilation. However, there are several drawbacks, such as portability between different simulators and controllability over the injection places.

In this paper, we introduce a Verilog-based FI technique overcoming the drawbacks of these SC and CM techniques by utilizing the VPI. Our literature review shows that there is no other report on a VPI-based FI technique. In order to ease discussions, we use the following terminologies in the subsequent sections and comply with [18]. A *fault* is a deviation in a hardware or software component from its intended function. *Faults* can be categorized into *permanent* and *transient* faults by their duration. An *error* is the manifestation of a *fault* on the observed interfaces. A *failure* is defined as the deviation of the delivered service from the specified service.

3. Fault Models

In order to make fault injection and evaluation feasible, the fault models used in this work refer to single bit faults. A variety of fault models are defined in order to represent real physical faults that occur in integrated circuits (ICs) [2], [6]. Table 2 lists the fault types initially implemented in this work. $N(t)$ is the original value, while $F(t)$ is the value of injected fault. t_s is the time of injection instant, and t_e is the time when a fault ends. Permanent faults can be modeled by setting t_e to the end time of the simulation. In this work, the *bit-flip* representing the inverted value at the instant t_s is differentiated from the *toggling-bit-flip* that toggles with original value. While the table only shows a typical set of basic faults, it was a dedicated objective to develop a flexible platform, that can be extended with additional fault types easily.

Table 2: Initially targeted fault types

Fault Type	*Fault Value Expression* $(t_s \leq t \leq t_e)$
Stuck-at 0	$F(t) = 0$
Stuck-at 1	$F(t) = 1$
Indetermination	$F(t) = X$
High Impedance	$F(t) = Z$
Bit-flip	$F(t) = not(N(t_s))$
Toggling-bit-flip	$F(t) = not(N(t))$

4. VPI based Fault Injection

As introduced in Section 2, three essential techniques for simulated FI are *saboteurs*, *mutants* and *simulator commands*. *Saboteurs* and *mutants* require source code modification resulting in recompilation which can be crucial for the overall runtime depending on the number, variety and size of experiments. However, there are no restrictions on the choice of the simulator. The SC technique does not require recompilation but is simulator dependent resulting in limited portability between different simulators. In principle, this FI technique could be applied to simulated designs independently from the used HDL. However, due to certain standardized behavior of Verilog, a fault cannot be injected at every place using SCs, which is illustrated with the mechanism of event driven simulation in Verilog in the following.

Essentially, there are six distinct regions of events in Verilog simulators for the current simulation time,

which are referred to as slots, as shown in Table 3 [19], [20]. Each slot includes several types of events. The events are executed successively from slot 1 to 6. SCs are placed into slot 1 to be executed. Thus, assuming that a fault is injected at a certain time and place, this event is scheduled in slot 1. If there are no events scheduled on the same place in other slots, the injected fault takes effect and the value can be held till the next simulation time step. However, if there are events scheduled after slot 1 at the same place, the injected fault will be overwritten and never take effect.

Table 3: Stratified event queue in Verilog simulators

Slot: Queue	*Events*
1: Active events queue	Blocking assignments
	Evaluate right-hand side of nonblocking assignments
	Continuous assignment
	$display and $write execution
	Evaluate inputs and change outputs of primitives
	PLI calltf routines
2: Inactive events queue	#0 blocking assignment
3: Nonblocking events queue	Update left-hand side of nonblocking assignments
4: PLI read-write synchronization	Registered cbReadWriteSynch simulation callbacks
5: Monitor events queue	$monitor command execution
	$strobe command execution
6: PLI read-only synchronization	Registered cbReadOnlySynch simulation callbacks

The Verilog HDL contains two types of procedural assignment statements, i.e the *blocking* and the *nonblocking procedural assignment* [21], which are used for updating the potential injection places. For instance, if there is a *nonblocking assignment* assigning values to the place the same as the injection place, the injected value will be overwritten by the *nonblocking assignment* scheduled in slot 3, as shown in Table 3. This example shows, that it is impossible to inject a fault on places where a *nonblocking assignment* is used by applying the Verilog-based *simulator commands* technique. Therefore, we introduce a novel injection technique, overcoming the limits of the *simulator commands* technique for Verilog simulation by accessing the last slot of a simulation time step (i.e. slot 6). Access to slot 6 of the Verilog event queue is enabled by the Verilog programming interface (VPI) [19] allowing to integrate C/C++ functions into a commercial Verilog simulator.

One important feature of the VPI is the ability to schedule and invoke specific simulation events, with which the *VPI-based* fault injection technique can be implemented. Figure 1 depicts the block diagram of this technique. Due to the different injection mechanisms of fault types, one callback function is created for each fault type. A single system task/function (TF) is provided to the user as an interface.

$HDL_InjectFault (fault type, place, instant, duration)

As this system TF needs to be called once for each fault to be injected, a testbench wrapper is created, in which the original testbench is instantiated. The system TF is called in the *initial* block of the wrapper specifying each time the four fault properties *fault type*, *place*, *instant* and *duration*. This way, different experiments can be driven by changing the testbench wrapper only, avoiding time consuming recompilation of the actual design.

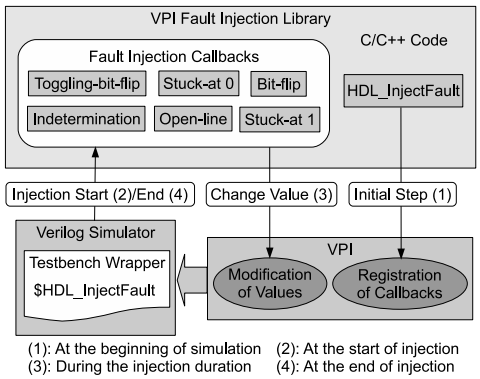

Figure 1: Implementation of VPI-based fault injection

At the beginning of the simulation (step 1), the simulator calls the system TF *$HDL_InjectFault*, in which one fault injection callback is registered according to the specified arguments *fault type* and *place*. This callback is to be executed at the injection instant and the end of injection, which are specified by other two arguments, *instant* and *duration*. Then, the simulator proceeds to the inject instant (step 2) and the according callback is executed. This callback sets the conditions when the value modification on the specified *place* will occur. During the injection (step 3), the value of the injection place is modified by using VPI to control the Verilog simulator. At the end of fault (step 4), the injection callback is executed again to restore the normal value. Several important features of different HDL-based injection techniques are listed in Table 4. Compared with the *simulator-commands* technique, the limitation on the injection place is removed with the *VPI-based* technique. In addition, since the VPI is IEEE standard, it can be applied to all VPI compliant Verilog simulators, while the

Table 4: Comparison of HDL-based fault injection techniques

Injection Technique	*Code Modification & Recompilation*	*Simulators*	*Verilog / VHDL*	*Injection Place*
Saboteur & Mutant	required	all	both	all
Simulator Commands	—	dedicated	both	most
VPI-based	—	VPI compliant simulators	Verilog	all

simulator-commands technique is simulator dedicated. Compared with the *saboteur* and *mutant* techniques, the *VPI-based* technique does not need to modify and re-compile the source code. However, it can only be applied to Verilog and VPI compliant simulators. Nevertheless, as soon as the VHDL programming interface (VHPI) is available [22], a similar VHDL-based injection technique can be developed. In section 6 we will show that the framework already takes this into account by being extendable for other HDLs.

5. Failure Classification and Evaluation

The following error related metrics are defined to evaluate the impact of the injected faults.

Error manifestation rate (EMR): The EMR is the percentage of experiments which indicate errors on the interface of a device. $EMR = N_e/N_i$, where N_e indicates the number of experiments with manifested errors on the interface of a device, and N_i is the number of FI experiments. Errors manifest when the injected faults cause inequality on the interface.

Error propagation latency (EPL): The EPL is defined as the time required for the injected fault to reach the interface of a device. $EPL = t_p - t_{inj}$, where t_{inj} is the time when the fault is injected, and t_p is the instant when the first error manifests on the device interface.

The impact of errors manifested on the interface of a device is further analyzed at the application level and categorized into *failures*, which show deviation of the delivered service from the specified one.

The classification of failures depends on the delivered service of a device. Principally, failures can be user-defined according to the individual device. For evaluation of our concept, we focus on a failure classification of a processor as shown in Table 5. The proposed framework, however, can be easily extended with other classifications of failures.

Table 5: Failure types for a processor core

Failure Type	*Definition*
Crash	The accessed memory location is out of boundaries of the application image
Data violation	The program terminates successfully. However, the memory content is different from that of the golden run.
Timeout	The program does not complete in the expected time t_s. ($t_s = t_n \times (1+10\%)$, where t_n is the normal execution time without fault injection)
Complete with delay	The program completes with delay in the expected time
Error without effect	In spite of the errors on the interface, the program terminates correctly

The FI evaluation is comprised of both the error evaluation on the processor interface and the failure evaluation at the application level. According to traditional evaluation approaches [2], [6], [8], FI evaluation is based on the simulation traces recorded during each FI simulation. After all the simulations are finished, the traces of processor interface in the FI simulation are compared with that of golden run for the further analysis. Obviously this approach has two drawbacks. On the one hand, many simulations are required to ensure a convincing statistical analysis. Therefore, the consumption of disk space is significant especially for large test cases. On the other hand, the dump of simulation traces consumes extra time.

Therefore, an on-the-fly evaluation approach is proposed in this work. Instead of dumping the traces in each FI simulation, only the traces of golden run are initially dumped. Then, during each fault simulation the traces of golden run are read on the fly and compared with the actual values of the interfaces using the *VPI*. If there are no inequalities detected, it means that no errors manifested on the interface. Conversely, if mismatches are found, their properties including place, time and value are dumped for further analysis. Several system TFs are created utilizing the VPI for the on-the-fly evaluation in a similar fashion as for FI.

Failure evaluation sets the injected faults into relation of their impact. The characteristics of faults include *injection place (P)*, *fault type (T)*, *fault duration (D)*, *injection instant (I)* and *possibility distribution function (PDF)*. Several metrics are defined to quantify the impact of injected faults. They include *EMR*, *EPL* and *failure distribution (FD)*.

In order to ease the result analysis, a *result analyzer* has been developed comprising several Matlab functions. Each function plots one characteristic of the fault space $\mathbb{F}\{P, T, D, I, PDF\}$ against one in the evaluation space $\mathbb{E}\{EMR, EPL, FD\}$.

6. A Flexible Platform for Fault Injection and Evaluation

Figure 2 shows an overview of the developed platform for fault injection and evaluation. Basically, the flow consists of three phases, i.e. setup, simulation and evaluation.

In the *setup phase*, the main objective is the generation of faults, which is controlled by a configuration file. This file is specified in XML format, and users can setup fault properties including all characteristics of the fault space $\mathbb{F}$. The failure types can be selected from the ones implemented in the *fault injection VPI library*. If necessary, this library can be extended with new fault types easily. A Verilog parser reads in the model of the target design and creates a *hardware intermediate representation (HWIR)*. This *HWIR* is independent from the used HDL and therefore allows for later extension to other HDLs by adding for instance a VHDL parser. The *fault generator* takes the configuration file and the *HWIR* as input to generate the simulation control script and the testbench wrapper. Two fault injection backends (BEs) are currently included: one for *simulator commands* and one for *VPI-based* FI. New FI techniques could be integrated easily by adding the according backend to the *fault generator*.

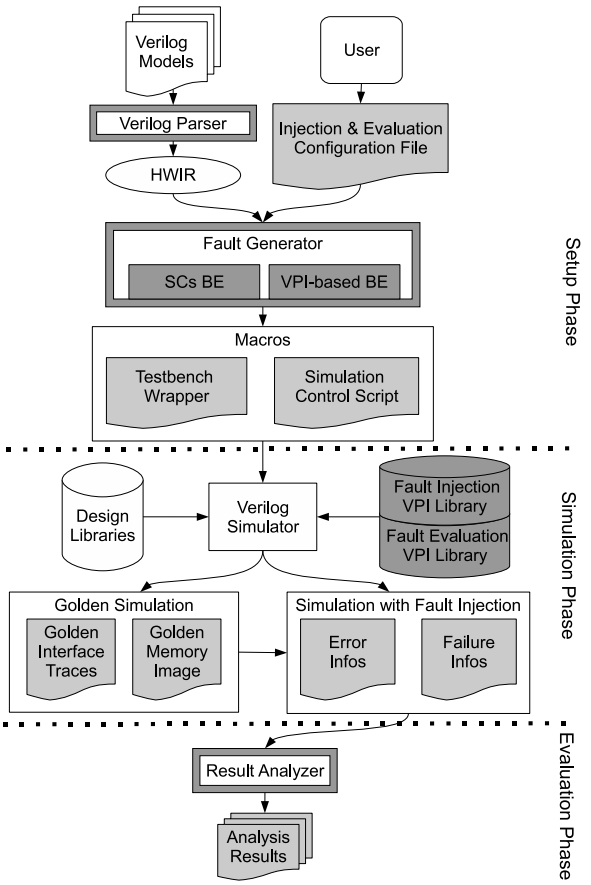

Figure 2: Overview of the fault injection and evaluation platform

In the *simulation phase*, two operations are carried out. Firstly, one golden run is performed to dump the traces of interface signals and the memory image. Then, the FI simulations are performed. According to the on-the-fly evaluation, the differences between traces of golden run and that of FI simulation are dumped directly to the error information and failure information files. In this phase, the created VPI libraries of fault injection and evaluation are linked to the simulator to enable the *VPI-based* FI and on-the-fly evaluation respectively. The libraries can be easily extended to support new fault types and even VHDL, once the VHPI is available.

Eventually, the dumped error and failure information are used in the *evaluation phase* by the *result analyzer* to create plots of dedicated relations between elements of the fault space $\mathbb{F}\{P, T, D, I, PDF\}$ and the evaluation space $\mathbb{E}\{EMR, EPL, FD\}$.

7. Example Case Study

In order to show the applicability and functionality of the platform, a case study is carried out for an application-specific instruction-set processor (ASIP), the ICORE [23], which is based on a mainly conventional DSP instruction set of a typical load/store Harvard-DSP architecture and extended for DVB-T algorithms. The design has been synthesized with Synopsys Design Compiler using a $130nm$ standard cell library with a supply voltage of $(1.2V)$. A statistical analysis is carried out at both RTL and gate level. The CORDIC algorithm is chosen as application (259 cycles), because the ICORE contains special instructions speeding up its execution.

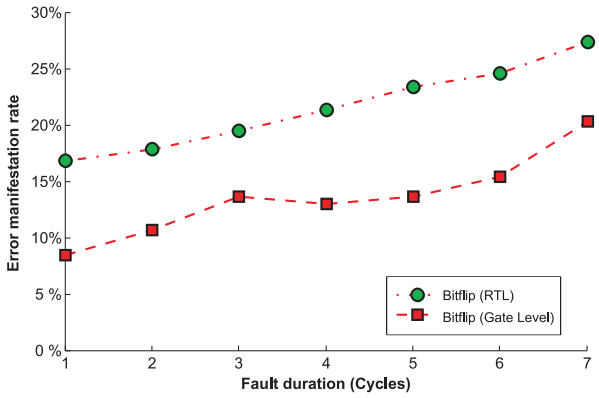

Figure 3: Exemplary error manifestation rate (random places over whole core)

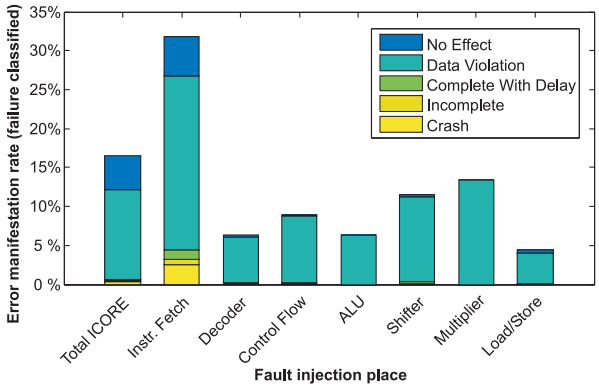

Figure 4: Exemplary failure distribution (1 cycle Bitflip, random place in according unit)

Figure 3 and 4 show two exemplary plots produced with our framework. Each measurement point results from 3000 FI experiments. Random selection of injection instance and place are equally distributed over whole simulation time and the architecture. The EMR shows similar growth with fault duration at RTL and gate level. However, absolute values differ meaning that it can make sense to do relative comparisons on RTL but absolute values are more accurate on gate level. Plotting the failure distribution normalized to the number of injected faults over the injection place helps to find fault susceptible units of an architecture (here, it can be seen that the fetch unit is much more susceptible than for instance the ALU).

Setting up FI experiments with our framework is quite comfortable and a matter of minutes. The simulation runtime depends heavily on the test case

and the host machine. However, injecting 1000 faults with our VPI-based FI into the gate-level simulation of the ICORE increases simulation runtime by 13% only. The same number of faults injected with SC-based FI results in a 82% longer simulation time. As our framework does not include code modifying techniques we cannot give a direct comparison. However, it was shown in [9] that these techniques do not result in shorter simulation times than SC-based FI.

8. Conclusion and Outlook

In this work, a novel *VPI-based* fault injection for Verilog simulations has been introduced. The VPI is also used for fast on-the-fly evaluation of simulations avoiding large simulation dump files. Based on these two techniques, a flexible and extendable platform has been developed. It supports both RTL and gate level. The feasibility and functionality of the platform has been shown by carrying out a number of injection and evaluation experiments using various injection configurations in a case study.

Future plans include the extension of fault models and failure types. Parallel simulation of experiments on multiple hosts to accelerate the statistical analysis is another focus for future research. Moreover, the concept of fault injection and evaluation using an HDL programming interface can be applied to VHDL with little effort, once the VHPI is available.

References

[1] S. Borkar *et al.*, "Parameter variations and impact on circuits and microarchitecture," in *Proc. 40th Design Automation Conf. (DAC 03)*, 2003, pp. 338–342.

[2] E. Jenn *et al.*, "Fault injection into VHDL models: the MEFISTO tool," in *Proc. 24th Int'l Symp. Fault-Tolerant Computing (FTCS 94)*, 1994, pp. 66–75.

[3] J. Boué, P. Pétillon, and Y. Crouzet, "MEFISTO-L: A VHDL-based fault injection tool for the experimental assessment of fault tolerance," in *Proc. 28th Ann. Int'l Symp. Fault-Tolerant Computing (FTCS 98)*, 1998, pp. 168–173.

[4] K. K. Goswami, R. K. Iyer, and L. Young, "DEPEND: A simulation-based environment for system level dependability analysis," *IEEE Trans. Computers*, vol. 46, no. 1, pp. 60–74, 1997.

[5] V. Sieh, O. Tschäche, and F. Balbach, "VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions," in *Proc. 27th Int'l Symp. Fault-Tolerant Computing (FTCS 97)*, 1997, pp. 32–36.

[6] J. C. Baraza *et al.*, "A prototype of a VHDL-based fault injection tool: description and application," *J. Systems Architecture*, vol. 47, no. 10, pp. 847–867, 2002.

[7] H. R. Zarandi, S. G. Miremadi, and A. Ejlali, "Fault injection into Verilog models for dependability evaluation of digital systems," in *Proc. 2nd Int'l Symp. Parallel and Distributed Computing*, 2003, pp. 281–287.

[8] H. R. Zarandi, S. G. Miremadi, and A. Ejlali, "Dependability analysis using a fault injection tool based on synthesizability of HDL models," in *Proc. 18th IEEE Int'l Symp. Defect and Fault Tolerance in VLSI Systems (DFT 03)*, 2003, pp. 485–492.

[9] J. C. Baraza *et al.*, "Improvement of fault injection techniques based on VHDL code modification," in *Proc. 10th IEEE Int'l High-Level Design Validation and Test Workshop (HLDVT 05)*, 2005, pp. 19–26.

[10] J. Arlat *et al.*, "Comparison of physical and software-implemented fault injection techniques," *IEEE Trans. Computers*, vol. 52, no. 9, pp. 1115–1133, 2003.

[11] J. Clark and D. Pradhan, "Fault injection: a method for validating computer-system dependability," *IEEE Computer*, vol. 28, no. 6, pp. 47–56, 1995.

[12] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," in *Proc. 19th Ann. Int'l Symp. Fault-Tolerant Computing (FTCS 89)*, 1989, pp. 340–347.

[13] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[14] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A tool for the validation of system dependability properties," in *Proc. 22nd Ann. Int'l Symp. Fault-Tolerant Computing (FTCS 92)*, 1992, pp. 336–344.

[15] T. K. Tsai and R. K. Iyer, "An approach to benchmarking of fault-tolerant commercial systems," in *Proc. 26th Ann. Int'l Symp. Fault-Tolerant Computing (FTCS 96)*, 1996, pp. 314–323.

[16] S. Han, K. G. Shin, and H. A. Rosenberg, "DOCTOR: An integrated software fault injection environment for distributed real-time systems," in *Proc. Int'l Computer Performance and Dependability Symp.*, 1995, pp. 204–213.

[17] T. A. Delong, B. W. Johnson, and J. A. Profeta III, "A fault injection technique for VHDL behavioral-level models," *IEEE Design & Test of Computers*, vol. 13, no. 4, pp. 24–33, 1996.

[18] J.-C. Laprie, "Dependable computing and fault tolerance: Concepts and terminology," in *Proc. 15th IEEE Intl Symp. Fault-Tolerant Computing (FTCS 85)*, 1985, pp. 2–11.

[19] S. Sutherland, *The Verilog PLI Handbook, Second Edition*. Kluwer Academic Publishers, 2002.

[20] C. E. Cummings, "Nonblocking assignments in Verilog synthesis, coding styles that kill!" in *Proc. Synopsys Users Group Conf. (SNUG 00)*, 2000.

[21] *IEEE Std 1364-2001 Verilog Hardware Description Language*. IEEE Press, 2001.

[22] "IEEE DASC VHDL PLI task force," Apr. 2009. http://www.vhdl.org/vhdlpli/

[23] T. Glökler and S. Bitterlich and H. Meyr, "ICORE: A low-power application specific instruction set processor for DVB-T acquisition and tracking," in *Proc. 13th Ann. IEEE Int'l ASIC/SOC Conference*, 2000, pp. 102–106.