IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 1 Issue 9, November 2014.
www.ijiset.com
ISSN 2348 – 7968

Enhancement of Fault Injection Techniques Using Saboteurs and Mutants for Modification of VHDL Code

G.Praveen Santhosh Kumar[#1] T.Malathi[#2] P.Sankar[#3] S.Saravanan[#4]

[#1]Assistant Professor,[# 2]Assistant Professor, [#3]Assistant Professor, [#4]Professor
[#1,2,3,4]Muthayammal Engineering College, Rasipuram, Tamil Nadu, India

Abstract — **Various fault modeling methods have been proposed for tackling the problem of increasing test-data volume of contemporary. The test pattern generation by random fault injection does not produce efficient results. So, the proper fault modeling becomes important since the deep sub micrometer devices are expected to be increasingly sensitive to physical faults. For this reason, fault-tolerance mechanisms are more and more required in VLSI circuits. So, validating their dependability is a prior concern in the design process. fault injection techniques based on the use of hardware description languages offer important advantages with regard to other techniques. First, as this type of techniques can be applied during the design phase of the system, they permit reducing the time-to-market. Second, they present high controllability and reach ability. Among the different techniques, those based on the use of saboteurs and mutants are especially attractive due to their high fault modeling capability. However, implementing automatically these techniques in a fault injection tool is difficult. Especially complex are the insertion of saboteurs and the generation of mutants. In this paper, we present new proposals to implement saboteurs and mutants for models in VHDL which are easy-to-automate, and whose philosophy can be generalized to other hardware description languages.**

Keywords— ***Hardware Implemented Fault Injection (HWIFI), single event upsets (SEUs), Unidirectional Serial Saboteur (USS), N -Bit Unidirectional Serial Saboteur (nUSS)***

I. INTRODUCTION

The new deep sub micrometer technologies are increasingly sensitive to physical faults, both to those due to external phenomena (i.e., transient faults such as single event upsets (SEUs), single event transient (SETs), etc.) and to internal defects (i.e., intermittent and permanent faults). Moreover, this sensitivity implies not only a raise of the fault rate, but also an increment of the likelihood of appearing multiple faults [1]–[3]. For this reason, the dependability of systems must be analyzed. This analysis can be either the study of the incidence of faults on the system (called *error syndrome* analysis) or checking the design specifications (called *validation*). The objective of the error syndrome analysis is to detect those parts of the system which are most sensitive to faults, and eventually, to choose the most suitable fault-tolerance mechanisms (FTMs). The aim of the validation is to verify that the system and/or its built-in FTMs accomplish the design specifications in presence of faults. If the dependability is analyzed at early phases of the design cycle, both time and money can be saved in the development process. A common experimental method to validate the dependability of a fault tolerant system (FTS) is fault injection, which is defined in [4] as *the deliberate introduction of faults into a system (the target system)*. Fault injection techniques can be classified in three main categories [5]: physical [also known as Hardware Implemented Fault Injection (HWIFI)], software implemented (SWIFI), and simulation-based. HWIFI is accomplished at physical level, disturbing the hardware with parameters of the environment (heavy ions radiation, electromagnetic interferences, etc.) or modifying the logic value of the pins of the integrated circuits. The objective of SWIFI consists of reproducing at software level the errors that would have been produced upon occurring faults in the hardware or the software. In simulation-based fault injection, the system under test is simulated in another computer system. The faults are induced altering the logical values during the simulation. Simulation-based fault injection is a useful experimental way to evaluate the dependability of a system during the design phase, thus reducing the time-to-market [6]–[8]. Another interesting advantage of this group of techniques with regard to others is that those based on simulation offer both high observability and controllability of all the modeled components [9]. Particularly, there exist a group of fault injection techniques based on the use of hardware description languages (or HDLs) as modeling languages. The most popular high-level HDLs are VHDL, Verilog, and SystemC. In our case, we work with VHDL [10].These techniques are widely applied, due to the advantages of employing an HDL. The present work is framed in this group of techniques. Fig. 1 shows a classification of VHDL-based fault injection techniques. Nevertheless, both this taxonomy and the description of the injection techniques can be generalized to

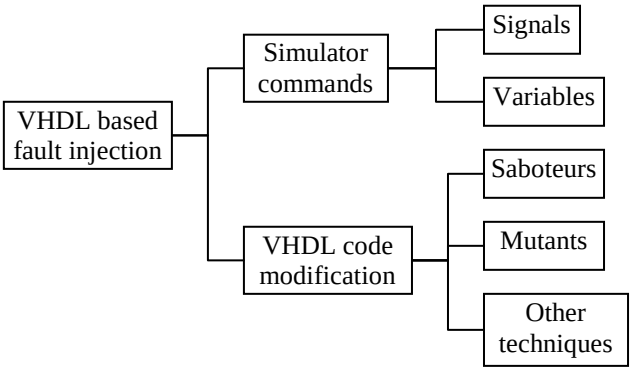

Figure.1.VHDL- based fault injection techniques

ISSN 2348 – 7968

any other HDL. *Simulator commands* technique is based on the use of simulator commands to modify the value or timing of the model signals and variables, without altering the VHDL code [6]. In the remaining techniques, the original VHDL code of the model is modified, either inserting *saboteurs* [6], [11], [12] or mutating the components of the model [6], [13], [14]. The techniques labeled as *Other techniques* are implemented by extending the VHDL language, either by adding new data types and signals, or modifying the VHDL resolution functions [7], [15]. The new data types and signals defined include the fault behavior description. However, these techniques require developing *ad hoc* compilers and simulators, and introducing control algorithms to manage the language extensions. There are works related to fault injection with saboteurs and mutants in other areas like test or field-programmable gate array (FPGA)-based fault emulation, although the objective of the study in each area is quite different. In dependability analysis, the objective can be either to verify the sensitivity to physical faults or validate the effectiveness of the FTMs of a simulation model (not necessarily synthesizable) of the system under analysis, by modifying the operation of the model at simulation time. In test, the aim of fault injection is to accelerate the test process by obtaining reduced test pattern lists injecting faults at higher abstraction levels, like register-transfer (RT) or system. For instance, in [16], a fault simulation tool has been developed for system models designed in Verilog at RT level. The purpose of this tool is not only to verify the model, but also to get the test pattern set that obtains the best correlation in the fault coverage between RT level and gate level. The RT fault simulator is based on simulating a modified version of the system model in which a number of zero-delay buffers (similar to *serial simple saboteurs*—see Section IV-A) are strategically inserted according to two statistical criteria: optimistic and pessimistic analysis. The modified model is then simulated with a commercial fault simulation tool called Verifault. Also, in [17], a fault simulation tool is developed, but, in this case, it accepts VHDL models at system level. Another important difference with the work in [16] is that the fault simulator developed performs the fault simulation by itself, instead of using a commercial fault simulator. Last, but not least, another important dissimilarity is the fact that the original model is *mutated* by inserting a special type of function able to alter the behavior of the system (see Section II-C for details). Finally, in [18], a technique to obtain the stratified coverage of a complex (that is, composed of multiple internal components) Verilog model at RT level is presented. Like in [16], Mao and Gulati use a gate-level commercial fault simulator (in this case, Verifault-XL) to simulate a modified version of the model in which a number of zero-delay buffers are judiciously inserted. In FPGA-based fault emulation, the objective of fault injection by using saboteurs and mutants is to synthesize into an FPGA a modified version of the original model that can be managed externally in order to emulate a faulty behavior. Interesting works in this area are [19], where mutants are implemented, [20] that applies saboteurs, and [21] that implements behavioral saboteurs. On the other hand, our research group has developed VFIT [22], [23], a VHDL-based fault injection tool that applies several of the previously described techniques. In fact, only the *other techniques* group has not been implemented due to their excessive complexity.

II. VHDL-BASED FAULT INJECTION TECHNIQUES

A. Fault Injection Using Simulator Commands

This fault injection technique is based on using the commands of the simulator at simulation time, in order to modify the value or timing of the signals and variables of the model [24]. Moreover, as VHDL generic constants are managed as special variables, it is possible to inject some non-usual fault models, such as delay faults [8]. Using simulator commands it is possible to inject transient, permanent, and intermittent faults. Though, there exists one restriction: due to the special nature of variables in VHDL, it is not possible to inject permanent faults in variables. This technique is the easiest one to implement and its temporal cost (to perform the simulation) is by far the lowest. However, the number of fault models that can be injected is smaller than with the other techniques.

B. Fault Injection with Saboteur

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

Attending to how saboteurs are inserted in the model, two types can be distinguished: serial and parallel [6]. As Fig.2(a) shows, a serial saboteur interposes between a component input port (*I* in the figure) and its source signal (*O* in the figure), whereas a parallel saboteur [see Fig. 2(b)] is added as an additional source (*S* in the figure) of a given signal. Parallel saboteurs have two important drawbacks respect to serial. First, implementing them is noticeably more complex because it is necessary to modify the data type of the signal affected, as well as the resolution function associated to the data type (a resolution function defines how values from multiple sources are resolved into a single value). Second, they allow to inject fewer fault models. For these reasons, their implementation has no special interest. So, in this paper, only serial saboteurs will be considered.

C. Fault Injection with Mutants

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

ISSN 2348 – 7968

TABLE 1
FAULT MODELS INJECTED BY VFIT AT GATE AND RT LEVELS

Injection techniques	Transient fault	Permanent/ intermittent faults
Simulator commands	pulse[a] , bit- flip[b], indetermination, delay	Stuck-at(0,1), indetermination, open-line, delay
Saboteurs	pulse[a],bit-flip[b],indetermination, delay	Stuck-at(0,1), indetermination, open-line, delay, short, Bridging, stuck-open
Mutants	Stuck-then, stuck-else, assignment control, dead process,dead clause, micro- operation, local stuck-data, Global stuck -data	Stuck-then,stuck-else,assignment control, dead process, dead clause, micro-operation, local stuck-data, global stuck-data

[a]Applied to Combinational Logic, It Represents A Single Event Transient (SET)
[b]Applied to Storage elements (register and memory), It Represents a Single Event Upset (SEU)

The mutation can be made in three ways:
• by adding saboteurs to structural model descriptions;
• by modifying structural descriptions replacing sub-components (i.e., a NAND gate can be replaced by a NOR gate);
• by modifying syntactical structures of behavioral descriptions.

There can exist lots of possible mutations in a VHDL model, so representative subsets of faults at logical and RT levels must be considered [13], [14], [25]: replacing the values of conditions in if and case statements (called *stuck-then*, *stuck-else*, *dead clause*, etc.), disturbing assignment statements (*assignment control*, *global stuck-data*, etc.), disturbing operators in expressions (*micro-operation*, *local stuck-data*), etc.
• *Stuck-Then*: Replacement of the if condition by true.
• *Stuck-Else*: Replacement of the if condition by false.
• *Assignment Control*: Disturbing an assignment operation.
• *Dead Process*: Elimination of the sensitivity list of a process.
• *Dead Clause*: Elimination of a clause in a case.
• *Micro-Operation*: Disturbing an operator.
• *Local Stuck-Data*: Disturbing the value of a variable, constant, or signal in an expression.

• *Global Stuck-Data*: Elimination of all value modifications Of a variable or signal in an architecture. Many of these fault models do not have a direct correspondence with physical faults, but they can show somehow an erroneous internal operation.

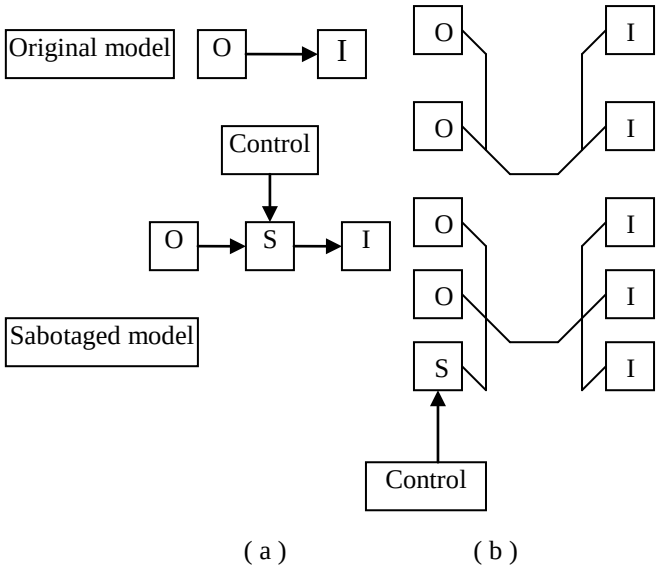

Figure. 2. Types of saboteurs (a) Serial. (b) Parallel

III. FAULT INJECTION ENVIRONMENT

The GSTF has developed a fault injection tool called VFIT (VHDL-based fault injection tool) [22], [23], that runs on PC computers (or compatible) under Windows and is model-independent. Although it admits models at any abstraction level, it has been mainly used on models at gate and RT levels. With VFIT it is possible to inject faults automatically applying the simulator commands technique. It is also feasible to inject faults using saboteurs and mutants, but in this case, the injection process needs the intervention of the user because the insertion of the saboteurs and the generation of mutants are not automatic. It can inject permanent, transient, and intermittent faults. When applied to models at gate and RT levels, it uses a wide set of fault models that try to be representative of deep sub micrometer technologies (see Table I). This set surpasses the classical stuck-at (for permanent faults) and bit-flip (for transient faults).

IV. AUTOMATING THE INSERTION OF SABOTEURS

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

A. Previous Models

VFIT can inject faults using serial saboteurs inserted manually in the design. The models of saboteurs implemented are as follows
• *Serial Simple Saboteur (SSS):* It interrupts the connection between an unidirectional local port of a component and its formal port, modifying either its value or its timing.

ISSN 2348 – 7968

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

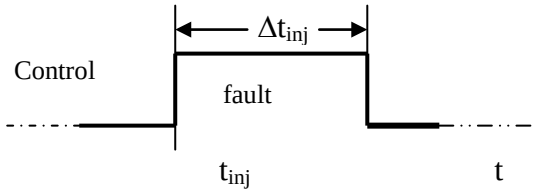

Figure.3. Timing of fault injection.

•*N -Bit Bidirectional Simple Saboteur (*nBSS *):* It is used with -bit bidirectional buses (for instance, data and control), and it is composed by SSBSs.
•*N -Bit Unidirectional Complex Saboteur (*nUCS *):* It applies to -bit unidirectional buses, and it is composed by SCSs.
•*N -Bit Bidirectional Complex Saboteur (nBCS):* It is used with -bit bidirectional buses and composed by SCBSs.
Every saboteur is controlled by means of the following three inputs.
• Control, whose mission is the timing of the injection: its activation determines both the injection instant (t_{inj}) and the fault duration (Δt_{inj}). It can be seen more clearly in
• Selection, that allows selecting the fault model to be injected.

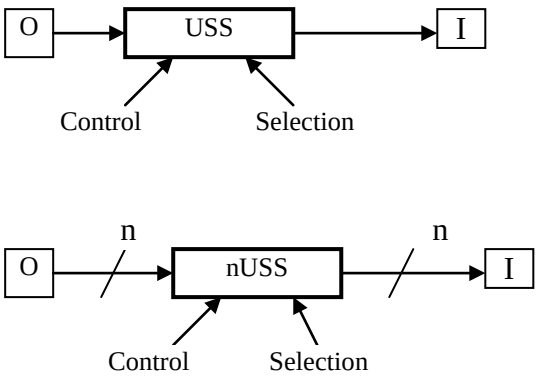

Figure. 4. USS and nUSS

B. Enhanced Models

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

• *Unidirectional Serial Saboteur (USS):* It is the same model as the SSS in the previous set, although the USS allows injecting new fault models.
•*N -Bit Unidirectional Serial Saboteur* (nUSS). This model replaces all the unidirectional multi-bit models in prior model set.
As the timing of Control and Selection inputs are identical, we have implemented an "optimized" version of these models in which the fault injection is managed only by Selection input. The idea is simple: when an injection is in progress, Selection indicates the fault(s) to be injected; but while no fault is injected, the value of Selection must represent a "no-fault" injection. However, this reduced version has a negative aspect: only single faults and multiple faults in the domain of time can be injected. To inject faults in the domain of space, the original scheme must be used.

C. Automatic Insertion of Saboteurs in the Design

The task of modifying automatically a source code seems apparently very complex. However, if the injection tool includes a parser, this is not actually so. From a syntactical tree of the model containing its complete structure, it is possible to go over the tree and generate a new copy of the source files, inserting new sentences or modifying other existing as needed. The insertion of saboteurs involves the following three actions:
1) Declaring the signals required to activate the saboteurs and to select the fault model to be injected;
2) Declaring the components of the saboteurs introduced;
3) Inserting the instances of the saboteurs, interposing between local and formal ports of the sabotaged components; this also implies declaring new signals to connect the saboteurs to local ports, and modifying the original mapping of ports.

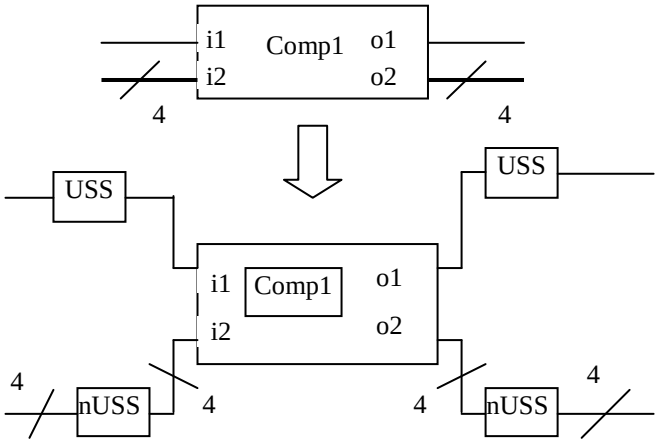

Figure. 5. Example of perturbation of a model

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

ISSN 2348 – 7968

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

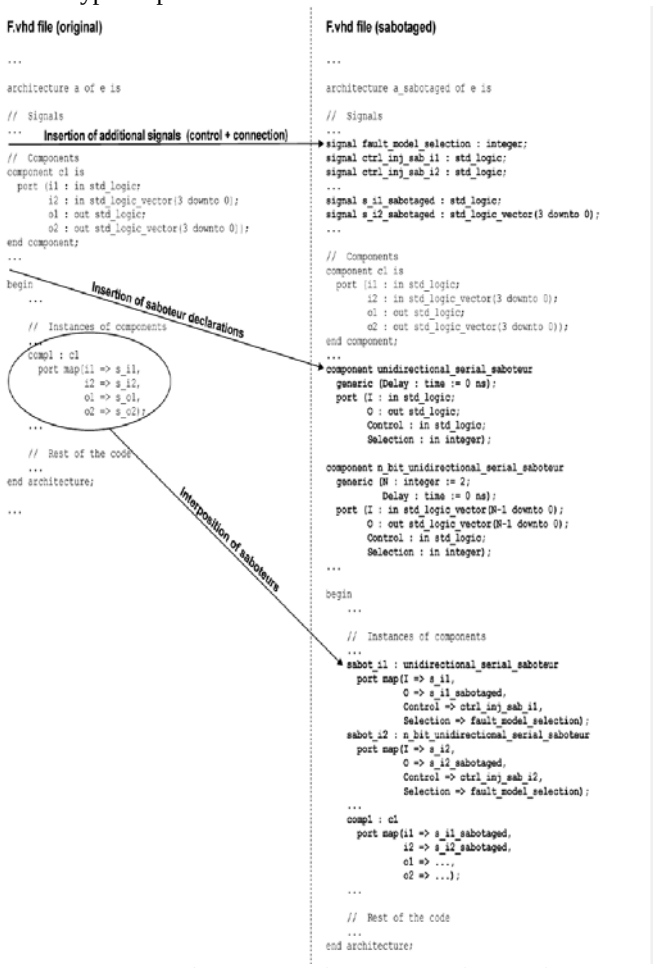

Figure. 6. Example of perturbation of a model. Modification of the VHDL code

V. AUTOMATING THE GENERATION OF MUTANTS

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

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

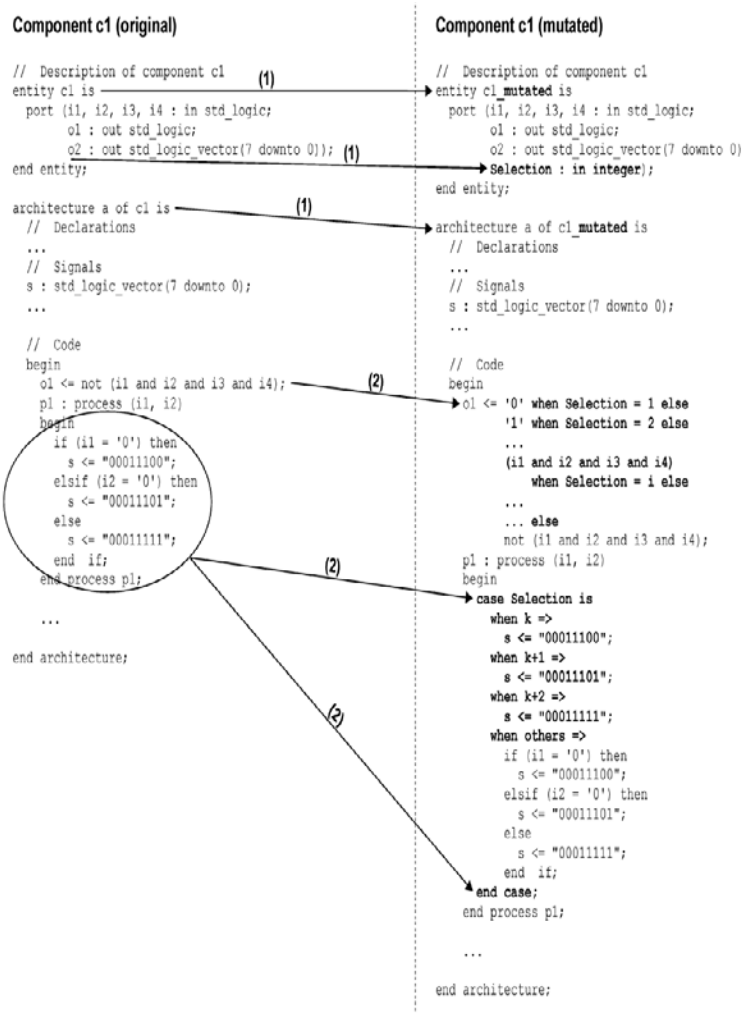

Figure. 7. Example of mutation of a model. Modification of the VHDL code of component

A. Previous Approaches

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

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

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

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

To fix this problem, a *dynamic* approach has been developed. It is based on the use of guarded signals together with the configuration mechanism [24]. In this way, at

ISSN 2348 – 7968

simulation time it is possible to stop the simulation of the original version of the model, and restart it simulating a faulty configuration. By using a number of simulator commands, the status of the simulation (that includes the simulation time and the value of all the signals and variables of the model) of the original version of the model can be saved on a file, and the same status is restored in the simulation of the faulty configuration. With the dynamic approach it is possible to inject (at any injection time) permanent, transient, and intermittent faults.

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

B. New Proposal to Implement Mutants

To avoid synchronizing simulations, we suggest a "brute force" implementation. What we propose is quite simple: to generate a unique mutated version of every architecture used in the model that includes all the possibilities of mutation considered previously in the setup phase [26].

In most cases, the modifications in the code are included by using if and case statements, although there are other possibilities, as shown in the example in Figs. 7. The aim of this type of modifications is to allow choosing among the correct statement and multiple wrong versions. For this purpose, a new input port (called Selection) must be inserted in the interface of the entity. The mission of Selection port is to specify the particular mutation to be activated, by asking its value in every "branch" of the mutated code. We call "branch" to every statement inserted to select between the correct operation and the wrong ones. The condition to activate one of the options is that the value of Selection coincides with the value specified in the
Selection statement. Another modification required is to declare a Fault Selection signal in the upper level of the model, which will be associated (mapped) to every local Selection port of the mutated components inserted, replacing the original ones. With this approach, also injecting faults becomes very easy. By using simulator commands, the value of Fault Selection signal can be modified at simulation time. In this way, it is possible to inject faults of the same time characteristics than with simulator-commands technique: transient, permanent, and intermittent.

C. Automatic Generation of Mutants

This new proposal to implement mutants is so simple that automating the generation of mutants of a given model is not complicated at all. Assuming that an injection tool has a parser, locating in the code the target statements to be mutated and replacing them with new ones is very easy. Next, we show a practical example. Fig. 7 represents the mutation of a component inserted in a given model. At the left side, we can see the original VHDL code of the component, and at the right side, the mutated code. Here, the text in bold types represents the modifications introduced. The arrows labeled with (1) correspond to modifications in the interface, and those labeled with (2) to the statements' mutation. In the example, a *signal assignment* and an if statement have been mutated. The *signal assignment* has been replaced with a *conditional signal assignment*, and to mutate the if, a case statement has been inserted. Both operations are relatively easy to perform automatically.

VI. SIMULATION RESULTS

A. TEST PATTERN GENERATION WITH SIMULATOR COMMANDS

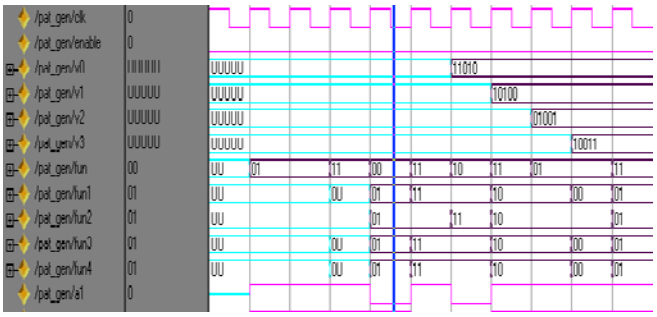

Figure 8 Test Pattern Generation with Simulator Commands

B. TEST PATTERN WITH MUTANTS

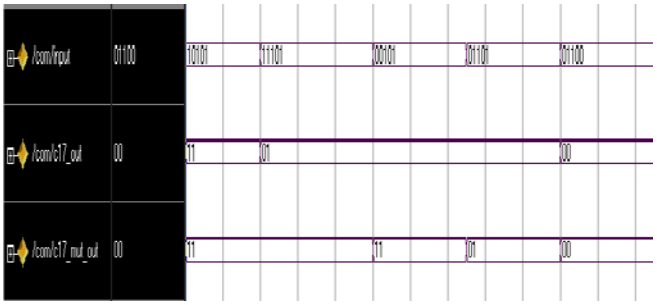

Figure 9 Test Pattern with Mutants

C. TEST PATTERN WITH SABOTEURS

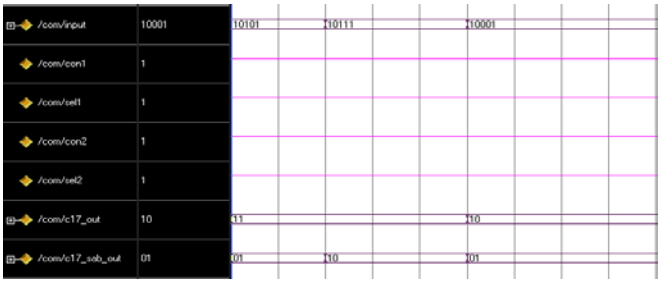

Figure 10 Test Pattern with Saboteurs

ISSN 2348 – 7968

D. TEST PATTERN GENERATION WITH SABOTEURS AND MUTANTS

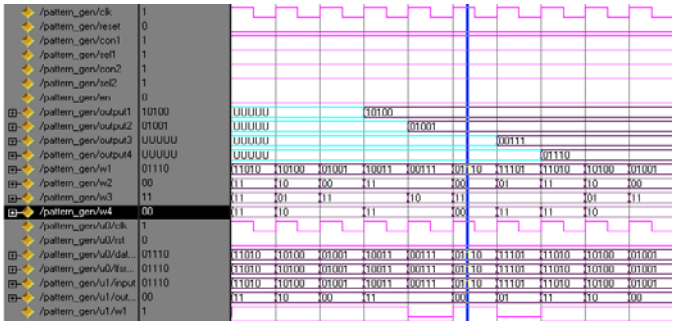

Figure 11 Test Pattern Generation with Saboteurs and Mutants

E. COMPARISON TABLE FOR FAULT COVERAGE

TABLE 2
FAULT COVERAGE

S.NO	Benchmark Circuit	Fault Coverage For Test Pattern Generated Using Simulator Command	Fault Coverage For Test Pattern Generated Using Saboteur & Mutant
1	C17	75%	87.5%

VII. CONCLUSION

This paper has presented a new VHDL based fault injection approach to fault modeling. A saboteur is a special VHDL component added to the original model. When activated, the mission of this component is to alter the value, or timing characteristics, of one or more signals, simulating the occurrence of a fault. Finally, a benchmark circuits were tested to demonstrate the strengths and limitations of the model. This would produce test patterns that specifically target the realistic logical faults and relevant delay faults that are neglected by other fault models. The advantages of the new proposal to implement mutants are especially relevant: it is easy to automate and reduces notably the spatial overhead.

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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