

Concurrent Hierarchical Fault Simulation using VHDL

Teresa Riesgo,
Universidad Politécnica de Madrid.
División de Ingeniería Electrónica
E.T.S. Ingenieros Industriales
c/ José Gutiérrez Abascal, 2.
Madrid, 28006 SPAIN.
Phone.: (+34) 1 4117517.
Fax: (+34) 1 5645966.

Serafin Olcoz,
Electronics Department.
Advance Technology Division.
TGI S.A.
Plaza Marqués Salamanca 3.
Madrid, 28006 SPAIN.
Phone.: (+34) 1 3964925.
Fax: (+34) 1 3964841.

Abstract.

This work presents a new approach to develop a concurrent hierarchical fault simulator using full VHDL. Our approach is based on the underlying model of VHDL. Faults are injected through VHDL perturbations of the elaborated model. Fault simulation is carried out on the sequential statements of the elaborated model, so a behavioral fault model can be used. Then, effects of faults can be followed and identified through the design hierarchy. This allows to compare models at different abstraction levels. Finally, the architecture of a concurrent and hierarchical fault simulator under development is presented.

1. Introduction.

The standard VHDL (IEEE-STD.1076-87) [1], is the most widely supported Hardware Description Language (HDL) by the industry, research centers and CAD vendors. This HDL covers a wide range of description levels (from system level to gate level) and supports different description styles (structural, data-flow and behavioral).

A new design methodology based on VHDL is emerging, supported by different tools covering most of the aspects of the electronic design cycle (simulation, synthesis, etc.).

Inside the design process there is an important and time consuming task: test. The designer must generate not only the circuit representation, but also the test vectors to verify that the circuit has no defect after fabrication. Thus, new test strategies must be proposed in order to have all the advantages of the VHDL based design methodology.

In this paper a new approach to fault simulation, which is one of the most important tools in the testing area, is presented. Advantages are taken from the VHDL features that make possible the use of hierarchy and concurrence in the fault simulation strategy. Section 2 introduces testing approaches, covering fault models and test tools. Section 3 presents the VHDL underlying model, in which we define the algorithmic fault model. This fault is the base of the VHDL fault simulator, shown in Section 4. Finally, Section 5 presents the conclusions of this work.

2. Testing Approaches.

Due to the nature of integrated circuits, test is one of the most critical tasks and special attention should be paid on it. In [2] and [3] good and precise explanations of all the topics related to testing problems and manufacturing test can be found. Besides production test (applied to the physical device, once it has been fabricated), design verification is necessary during the development phase, which establishes the correspondence between the final design (implementation) and the expected functionality (specification). Therefore, the test strategy has to be present during the whole development cycle.

Most of the classical test procedures have been applied at the logic gate or switch levels. However, when using higher abstraction levels, as allowed by a VHDL based design methodology, new test tools must be developed. In any case, a fault model and related tools, such as a fault simulator, are necessary in the testing tasks.

Following subsections show a description of the fault models and their evolution, as well as a summary of the main tools involved in the testing procedures.

2.1. Fault Models.

A fault model is an abstract representation of the effects that physical failures produce in the circuit behavior. The number and types of possible defects in a circuit is so large that it is impossible to evaluate or generate a set of test vectors to verify that the circuit has no fabrication defect. This is the reason why fault models are widely used and accepted as the reference for testing purposes.

A good fault model has to be simple enough to be easily used by both designers and CAD tools, and accurate enough to closely represent the failing behavior of the devices. A compromise between these two parameters has to be found.

Fault modeling is strongly related to:

- **The technology used**, depending on the circuit technology the possible defects are different. For example, the "single-stuck-at-0,1" fault model has been found to be ineffective for CMOS circuits as it does not clearly represents most of the realistic defects present in these circuits [4]. Another special case are some regular blocks, as PLAs or memories, whose fault models consider the special structure of the block. For example, for PLAs some fault models such as "cross-point" faults are usually considered.

- **The abstraction level considered**, different fault models may be used depending on the abstraction level considered. The most popular fault model is used at the logic gate level, which is the "single-stuck-at-0,1". However, if the circuit is modeled by a switch level structure is more accurate to use some other models, such as "stuck-open" and "stuck-on". When talking about higher level models, some new fault models are needed.

HDLs have the capability of describing the hardware system behavior in a technology independently way. This description can be made at different abstraction levels. Thus, the fault model used in a HDL based methodology is not constrained to the target technology and/or the chosen abstraction level.

A new kind of fault model, usually called behavioral fault model, is required. Besides, this fault model definition will depend on the used HDL, although the main idea is always similar because, in any case, faults are modeled as code perturbations.

Some HDL fault models have been proposed. In [5], C is used as hardware description language. They make an empirical comparison between the fault coverages at the behavioral level (with their own models), and at the gate level (with stuck-at models). Finally, they obtain a correspondence between both fault coverages, with a high correlation. Other authors, like [6], propose fault models for behavioral descriptions in VHDL. Their final goal is the development of an automatic test pattern generator for the model. The fault model is not compared with another model, defined at gate or switch level and it is very restrictive, both in the VHDL descriptions and in the fault model definition. Another approach to fault modeling for behavioral VHDL descriptions may be found in [7], being the considered descriptions those accepted by the current synthesis tools. This fault model considers not only the pre-synthesis description, but also the synthesized structure to accurately model faults in those blocks which are combinational logic, where stuck-at fault model at the logic level is used.

Next Section presents the fault model definition used in our approach. This fault model could be considered as behavioral, although it can be applied to any abstraction level and style of description allowed by VHDL.

2.2. Test Tools.

Some of the tasks related to the test generation and evaluation, during the design cycle, may be automatically

performed or, at least, aided by CAD tools. The generation of the test vectors may be made by Automatic Test Pattern Generators (ATPG). Well known algorithms exist for ATPGs, that work with combinational circuits (D, [8], SOCRATES, [9]). However, test generation for sequential circuits is much more difficult to perform automatically and some solutions coming from Design-For Testability techniques are used (Scan-Path, BIST, etc.), [10]. Test evaluation is performed by a fault simulator. A fault simulator computes the number of faults detected by a set of test vectors, what is to say, the fault coverage. Thus, it is clearly linked to the fault model used and the abstraction level of the input description. The information provided by a fault simulator may also be useful for detecting low testability points in a circuit, to help in the test generation tasks and even to verify the correctness of the functional test vectors. Some other test tools are used, as the testability measures, which statically (without simulation) computes the controllability and observability of each node of the circuit.

All these tools have been classically developed for logic level models and stuck-at fault models. However, in the last years some approaches consider higher level information for ATPGs, [6], because it provides to the tool a knowledge of the circuit very similar to the designer's and the solution for sequential ATPG may come from them.

In the case of fault simulation, it would be very useful to have a tool that computes the fault coverages at every design phase, independently of the abstraction level. This would be very useful, not only to help in the final test vectors generation but also in design error correction and verification.

VHDL provides a common platform to perform the fault simulation without the restriction of the abstraction level or the design stage.

3. VHDL Algorithmic Fault Model.

VHDL allows to describe a hardware system by means of a design hierarchy defining its structural and behavioral information at several levels. At the time of elaborating a simulable model, this design hierarchy is transformed into an heterarchy of VHDL processes, which only have behavioral information. Because every VHDL description has its corresponding elaborated model, it is only necessary to define an algorithmic fault model.

This fault model may be the base to make a fault simulation of any VHDL description, with the ability of carrying the resulting information to the original hierarchical description. Thus, this algorithmic fault model is not only applicable to a VHDL behavioral description but also to any hierarchical model.

3.1. VHDL a Processes based HDL.

A VHDL description produces an executable model through the elaboration task. This model is described by a set of communicating processes. The execution of this model simulates the behavior of the hardware system described by means of VHDL source code, [11]. Other HDL's (e.g., Verilog [12], and UDL/I, [13]) are based on the same underlying model, although each one has its own special features, i.e., timing model, event scheduling policy, etc. The executable model of VHDL may be useful for different applications, as it has been formalized using Coloured Petri Nets, [14]. This formal model is going to be used to analyze properties of a VHDL

description, and it is also an entry point to apply formal verification techniques, [15].

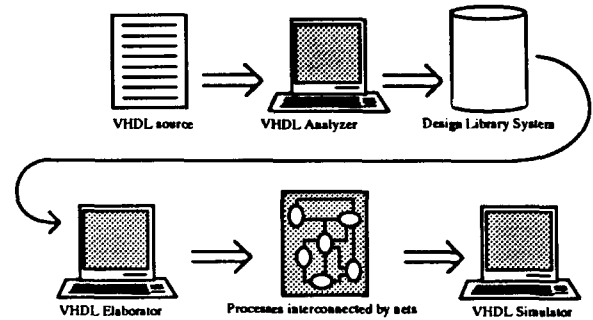


Figure 1.- Analysis, Elaboration and Execution of a VHDL description.

Following subsections detail the necessary steps to execute a VHDL description, see figure 1.

3.1.1. VHDL Analysis.

A VHDL description is a textual representation (ASCII) that follows the syntax rules of the VHDL Language Reference Manual (LRM) [1]. The textual description is stored in one or more text files, called design files. A design file is organized in one or more design units, that can be parsed in order to check the correctness of its syntax.

The VHDL Analyzer can be considered as a compiler that transforms a design file into a set of interconnected library units. Each library unit corresponds to each design unit of the design file. The context clauses of each design unit define the dependencies of the unit under analysis with other previously analyzed library units. The storage facility for library units, called design library, is implementation dependent. The output of the VHDL Analyzer does not create an executable model of a VHDL description, but creates a network of library units which allows their maintenance and management. The design library can be considered

as the interface between the design files and the VHDL based tools, such as a VHDL simulator or a synthesis tool.

3.1.2. VHDL Elaboration.

The elaboration is the previous step to execute a VHDL description. The elaboration task transforms a design hierarchy into a flat model consisting of VHDL processes interconnected by a network of signals, and their associated information. This model can be considered a heterarchy of processes because there is no hierarchy among the VHDL processes.

The elaborated model does not depend on the description style (behavioral, structural, data-flow, or mixed), chosen by the designer, neither on the abstraction level of the VHDL description. The elaborator saves the needed information to perform the backtracking, from the elaborated model to the VHDL source code.

In order to elaborate an executable model, the VHDL designer must choose the design entity to be simulated. This design entity is the top level entity of the design hierarchy describing the system under study. The designer can consider this entity as a workbench composed by a stimuli generator and the unit under test (UUT). This distinction is only in the designer mind because after the elaboration there is no difference between the processes coming from the UUT or from the stimuli generator.

Elaborated processes are composed by VHDL sequential statements, that are cyclically executed. Each process has its own state, represented by the value of the variables (local to the process) and the used signals (global to the process). Both, variable and signal values can be modified by means of

their corresponding assignment statements. Each process has also its own control flow that can be modified by means of: (a) *conditional* statements (if and case); (b) *iterative* statements (loops, both for and while in conjunction with next and exit); and (c) *wait* statements. When a process executes a wait statement suspends its execution flow until the conditions of the statement are satisfied and then, resumes its execution flow. A VHDL process can also contain assert statements, useful for simulation purposes, but without any influence in the model behavior. Although the elaborated model is flat, each process can be seen as a hierarchy of statements grouped in subprograms (procedures) like in any other structured programming language.

Each elaborated process is sensitive to events on signals, in other words, its execution is event-driven. Events arrive to a process through the signals and the process execution may also produce events on signals that are sent to the processes network. The way in which events are added to the queue, corresponding to the signal's driver, depends on the timing model used by the signal assignment statements of the processes.

Inertial and transport delay models are the two preemptive timing models available in VHDL. Both models have a different algorithm of adding events to the signal's driver.

The processes are asynchronously and concurrently executed by a VHDL simulator. The VHDL simulator is a part of the language definition given by the LRM. This simulator manages the time advance and the activity of the processes (event scheduling) during a simulation. The VHDL simulator is introduced in the LRM by means of an

abstract representation called kernel process.

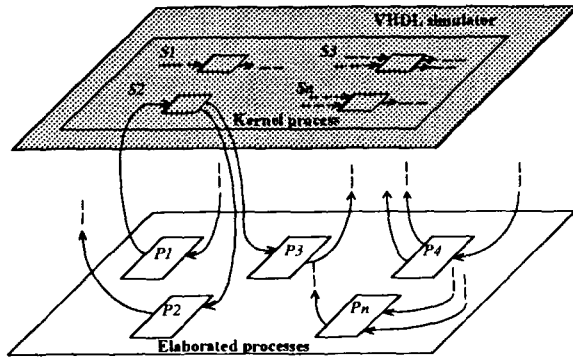


Figure 3.- Communicating processes model. We can consider the kernel process as a part of the heterarchy of processes, see figure 3. In this case,

communication between elaborated processes is not direct through signals any more, but it is carried out through global variables shared between the processes and the kernel.

3.1.3. VHDL Execution.

The execution of the model consists of an initialization phase followed by the repetitive execution of the sequential statements of each process, including the kernel process. The VHDL simulation finishes when Time'high is reached, see Figure 4.

During the initialization phase the kernel process is suspended until all processes reach a wait statement. Then, the first simulation cycle starts. Each repetition of the kernel process is

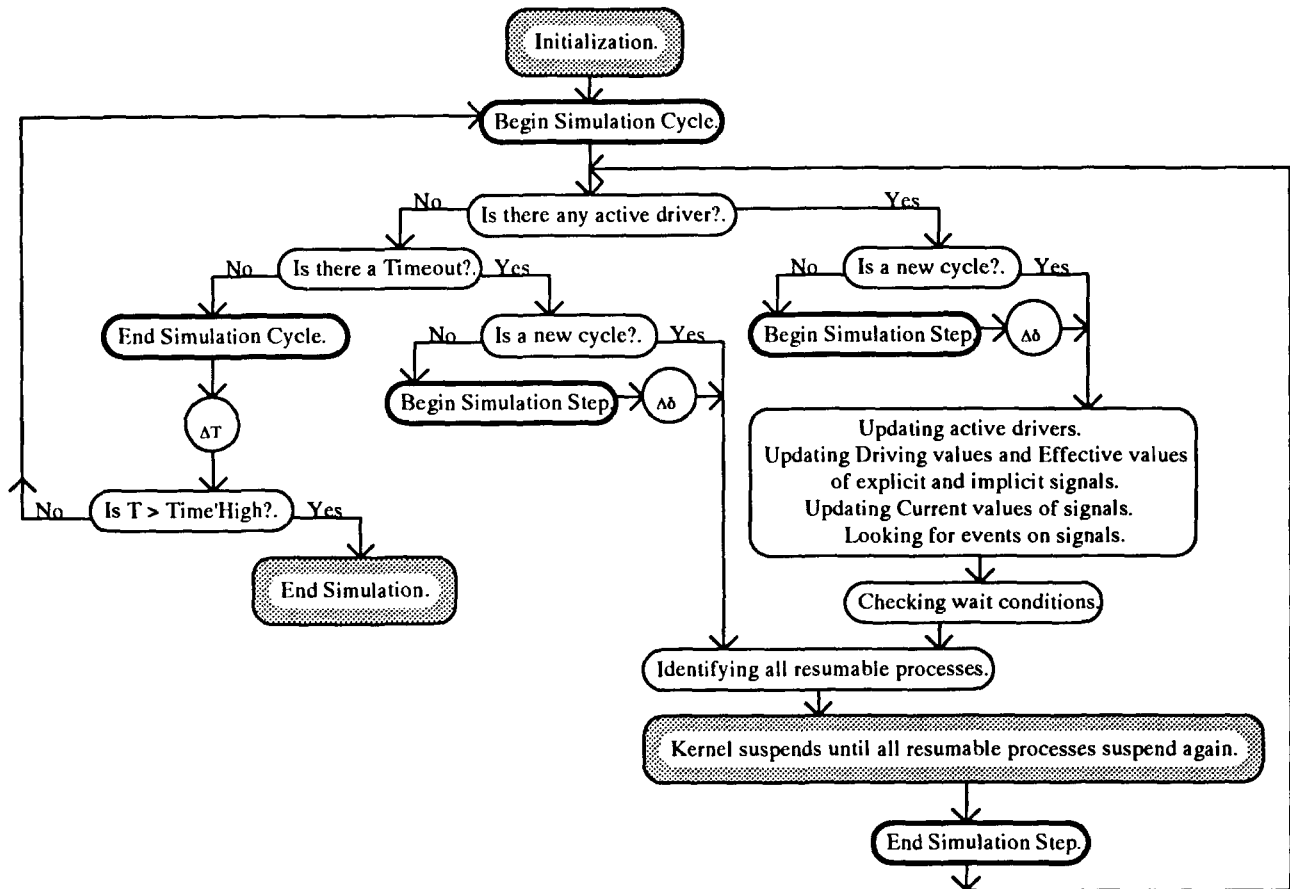


Figure 4.-Control flow of kernel process (VHDL simulation).

called a simulation cycle. In each cycle, the values of all signals are computed. If, as a result of this computation an event occurs on a given signal, processes that are sensitive to that signal will resume and will be executed as part of the simulation cycle; this is called a simulation step. So, the simulation cycle is a recursive one that can be compound by as many simulation steps as needed.

When the *wait* conditions of a process are satisfied, this process is added to the queue of processes to be executed during the current simulation cycle. When all executable processes for this simulation cycle have been identified, each process resumes its execution. Then, the kernel suspends until all these resumable processes reach a wait statement during this simulation step. Finally, the kernel process resumes the execution control when all processes are suspended again, and a new simulation step or simulation cycle starts.

A simulation step implies a change in the states of the processes without a timing advance ($\Delta\delta$). On the contrary, a new simulation cycle implies the increment the physical time (ΔT). With this complex control flow of the kernel process, VHDL offers a suitable timing model to describe any kind of hardware system.

3.1.4. VHDL Timing model.

The VHDL timing model was designed on the basis of a sub-unit delay model named δ (delta), see figure 5.

This sub-unit represents a basic simulation step, which has the feature of being next-step delay of a signal assignment, like in a unit-delay model.

But this step does not mean a unit of physical time as it is in pure unit-delay, and for this reason we can say that VHDL timing model is not pure

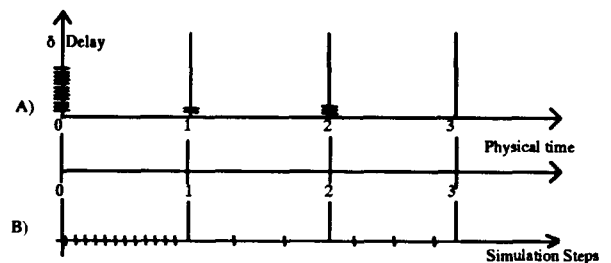


Figure 5 .- VHDL time scale:

A) Physical Time; B) Simulation steps.

unit-delay. The sub-unit delay model is also a feature of other HDLs such as BCL, [16], where it is named step.

On the other hand, the VHDL timing model allows modeling communication between processes without advancement of physical time units (in time less than unit). This feature is necessary to describe combinational hardware. If a VHDL description only uses the δ -delay model then the physical time never advances, representing only a causality relation.

A delta advancement of time is an infinitesimal, but not zero, advancement of simulation time. This is the difference between δ delay and classical zero-delay model.

Although δ -delay model allows communicating processes in time less than unit (1 fs) and modeling combinational hardware, is not the cause of independence of the processes execution order.

The execution result does not depend on the order in which elaborated processes are executed because VHDL 1076-87 does not allow direct communication between processes sharing variables.

Fault Classes	Affected Statements	Fault Model
Control Faults	Conditional: <i>if then else</i>	<i>stuck-then, stuck-else</i>
	Conditional: <i>case is when</i>	<i>dead-condition</i>
	Iterative: <i>for, while</i>	<i>dead-loop, dead-exit, dead-next</i>
	Function and Procedure call	<i>dead-call</i>
	Synchronization: <i>wait</i>	<i>dead-wait, stuck-true-condition, stuck-false-condition</i>
Data Faults	Signal or variable assignment: <i><=, :=</i>	<i>dead-assignment</i>
	Signals	<i>local-stuck-value</i> <i>global-stuck-value</i>
	Variables	<i>local-stuck-value</i> <i>global-stuck-value</i>

Table 1.- Summary of the VHDL fault model.

3.2. VHDL Fault Model.

The fault effect is the perturbation of the VHDL processes code. Therefore, the fault model is applied to algorithmic descriptions. Faults will change the data transfers, the control flow and the activation of processes. The VHDL fault model is classified into two classes, according to their effect in the description of the VHDL elaborated model: control faults and data faults.

Table 1 shows a summary of all kinds of faults, which are detailed in next subsection.

3.3. Detailed Description of the Fault Model

Control Faults: These faults affect the description changing the execution control flow of the model, making some statements are always or never executed independently of their condition.

•Conditionals

Stuck-then. The construction:
if condition then statement_1 else statement_2;

may fail and *statement_1* is always executed, while *statement_2* is never

executed. This is equivalent to say that the *condition* is set to "true".

Stuck-else. The construction:
if condition then statement_1 else statement_2;

may fail in the opposite way than *stuck-then* does, and *statement_2* is always executed, while *statement_1* is never executed. This is equivalent to say that the *condition* is set to "false". Several *elsif* may be nested in these constructions and the *stuck-else* fault model will affect to every *elsif* in the description.

Dead-condition: The *case* construction, whose syntax is
case expression is
when choices => statements;
otherwise => statements;

may fail making that one of the clauses never executes. The clauses are the *statements* that appear after the choice has been made. For the affected condition, the default statements (*otherwise*) will be executed instead of the appropriate one.

•Iteratives

Dead-loop. The loops are based on "for" and "while" structures like:
for (i in signal'range) loop statements end loop;

while (condition) loop statements end loop;

The fault associated to these loops affect the range of the counting variable *l* (*for*) converting it to zero, or to the *condition* (*while*) making it is always false.

Dead-exit This fault is also associated to already mentioned loops, and affects the range of the counting variable *l* (*for*) converting it to such a value that the loop never ends, or to the *condition* (*while*) making it is always true. If there exists an *exit* statement inside the loop the fault will make it never executes. The effect of this fault is that the loop will execute forever.

Dead-next: A *next* statement inside a loop may fail making it will never take place. The effect will be that regardless of the sentence, the loop will continue until it finishes.

●Subprograms

Dead-call. A procedure or a function call may appear in the VHDL code of an architecture. A *dead-call* fault will cause that the procedure or function call does not execute. When dealing with procedures, all the defined faults are applicable and have to be inserted. Inside a function, some faults will not be considered, as no signal assignment or *wait* statements are permitted by the language.

●Synchronization:

This subtype of faults considers those faults that directly affect the execution of the VHDL processes, suspending their execution or making them always active in the presence of a fault.

Stuck-wait. The construction:
wait (on signal) (until condition) (for time);

may fail making the process loses its *wait* statement. This fault will affect the synchronization scheme of the processes, making the process never be suspended or until the next *wait* statement in the process. If a process has only one *wait* , this fault will cause an error in the simulation. So, these processes have to be identified before the fault insertion procedure.

Stuck-true-condition. Besides making the *wait* statement does not take place, it is possible to consider that the fault is present in the condition of resumption of the process. In this case, the fault will cause that the process resumes at the next simulation step, because the condition is always true. This fault will be equivalent to switch the correct *wait* statement with:

wait for 0 ns;

Stuck-false-condition. The effect of this fault is the opposite of the previous one. In this case, the fault will cause that the process never resumes, because the condition is always false. This fault will be equivalent to switch the correct *wait* statement with:

wait;

Data Faults: These faults refer either to changes in the assigned values or to the assignment statements.

●Assignment

Dead-assignment. The construction for a signal or variable assignment statement may fail and never be executed. This could be equivalent to a *signal-(variable)-stuck-value* but the scope of the fault is restricted to this assignment. In the case of a signal assignment, the fault effect is multiple

because the fault can affect to multiple signal assignment.

•**Stuck-data.**

Global-stuck-value. This fault will make that a signal or a variable has a permanent value in its whole scope, from the initialization phase. The problem arises when the "value" to be "stuck to" is not clear, when the data is of a continuous type (i.e., real, integer). The chosen solution is that this kind of data may be stuck-to the bound values inside the signal or variable range and also two values that are often very significative: zero and one. For the enumerated types, all the possible stuck are considered.

Local-stuck-value. The effect of the fault is now restricted to a particular variable or signal assignment. The right side of the assignment statement is replaced by the stuck-data. In the signal assignment the scheduled time for the stuck-value will be 0 ns (by the fault model definition).

4.VHDL Fault Simulator.

The elaborated model of a VHDL description is the entry of the VHDL Fault Simulator (VFS). This description corresponds to a workbench including the Stimuli Generator and the UUT. For the VFS purpose the processes elaborated from the Stimuli Generator have to be identified because they will not be affected by the fault insertion.

Fault simulation can be divided in two different tools. the Fault Mapper (FM) and the VFS itself, see figure 6. In fact, the elaborated model is the entry of the FM, which generates the fault list based on the algorithmic fault model, and inserts these faults creating as many copies of the elaborated model as the number of existing faults.

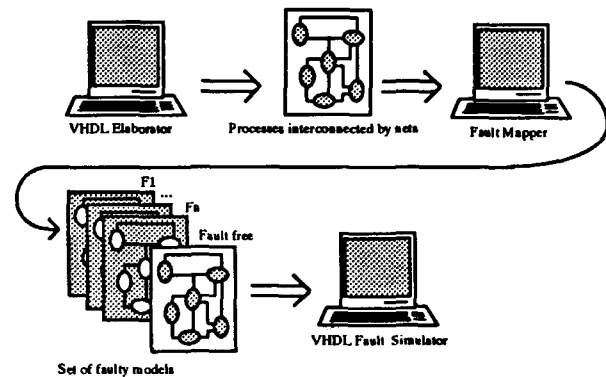


Figure 6.- Fault Simulator Architecture.

For large VHDL description the fault insertion must be partitioned into several subsets. In this case, every subset should have a fault free copy of the elaborated model as a reference in the final comparison.

The VFS has the same algorithms and control flow as a standard VHDL simulator (kernel process), but the VFS kernel has to deal with a set of faulty models instead of just one elaborated model. This implies that the VFS kernel must concurrently check for active drivers in the set of drivers corresponding to the faulty copies. In the same way it must compute the driving, effective and current values as well as the events, for every element of the set of faulty models. Then, the set of resumable processes is identified in order to perform the simulation.

The VHDL simulator provides the capability of observing the resulting values at each simulation step. The VFS will use this feature to compare the state of the processes (fault free vs. faulty copies) in each δ . A hierarchical analysis could be performed because the information of the design hierarchy is recoverable from the computing driving values part of the VFS.

A classical fault dropping technique, called fault dropping, is used to reduce

the number of faults to simulate dynamically. This technique consists on rejecting those faulty copies whose associated faults have been previously detected. A fault will be considered as detected when it produces a difference between the good and faulty descriptions in a simulation cycle (when simulation time is incremented in physical units). The result given by the VFS is the percentage of faults detected of the possible ones (algorithmic fault coverage), as well as a additional information such as the detected faults, the simulation time in which they were detected. This information is useful not only for testing but also for verification purposes.

Another application of this approach is the capability of comparing the faulty behavior of a hardware system described at different abstraction levels (i.e., pre and post synthesis) to refine the algorithmic fault model for a given technology or application. This can be made using two different elaborated models corresponding to the same description at different abstraction levels.

5. Conclusions.

A new approach to develop a concurrent hierarchical fault simulator using full VHDL has been presented. This approach is based on the underlying model of VHDL. An algorithmic fault model has been proposed that covers all behavioral features of the language. The fault effects can be followed and identified through the design hierarchy. This allows to compare models at different abstraction levels.

The algorithmic fault model will be updated when the IEEE publishes the new version of VHDL (1076-92).

Although some changes will affect to the elaborated model (i.e., shared variables, postponed processes, etc.), the main aspects of the fault model presented here will remain.

Using this fault model, with the elaborated model of a VHDL description, the architecture of a concurrent and hierarchical Fault Simulator (VFS) has been presented.

This work has not considered temporary faults (only permanent faults has been addressed), neither timing faults (which affect to signal assignment statements and timeout condition of wait statements). The complexity of the VFS would dramatically grow when considering timing or intermittent aspects in the fault model.

References.

- [1] *IEEE Standard VHDL Language Reference Manual*, IEEE, Inc., New York, N.Y., U.S.A., March 1988.
- [2] R. R. Fritzeimer, H. T. Nagle, C. F. Hawkins, "Fundamentals of Testability - A Tutorial", *IEEE Transactions on Industrial Electronics*, vol. 36, no. 2, May 1989.
- [3] C. F. Hawkins, H. T. Nagle, R. R. Fritzeimer, J. R. Guth, "The VLSI Circuit Test Problem - A Tutorial", *IEEE Transactions on Industrial Electronics*, vol. 36, no. 2, May 1989.
- [4] R. L. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits", *Bell System Technical Journal*, vol. 57, May-June 1978.
- [5] S. Ghosh, T. J. Chakraborty, "On Behavior Fault Modeling for Digital Designs", *Journal of Electronic Testing*:

Theory and Applications, vol. 2, pp. 135-151, 1991.

[6] J. R. Armstrong, F. Lam, and P. C. Ward, "Test Generation and Fault Simulation for Behavioral Models." In Joel M. Schoen (Ed.), *Performance and Fault Modeling with VHDL*, Prentice Hall, 1992, pp. 240-303.

[7] T. Riesgo, J. Uceda, "A Hardware Oriented Fault Model for Synthesizable Descriptions", *North Atlantic Test Workshop*, Hanover, N. H., June 1993.

[8] J. P. Roth, W. G. Bouricius, P. R. Schneider, "Programmed Algorithms to Compute Tests and Detect and Distinguish Between Failures in Logic Circuits", *IEEE Transactions on Electronic Computers*, Vol. EC-16, pp. 567-580, October 1967.

[9] M. Schluz, E. Trischler, T. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System", *IEEE Transactions on Computer-Aided Design*, Vol. CAD-7, no. 1, pp. 126-137., January 1988.

[10] T. W. Williams, K. P. Parker, "Design for Testability - A Survey", *Proceedings IEEE*, Vol. 71, pp. 98-112, January 1983.

[11] S. Olcoz, J. M. Colom, "VHDL Through the Looking Glass." *VHDL Forum for CAD in Europe*, March, Innsbruck 1993, pp. 13-22.

[12] Open Verilog International, *Verilog Hardware Description Language Reference Manual*. Version 1.0, October 1991.

[13] O. Karatsu, "VLSI Design Language Standardization Effort in Japan". *Proceedings of the 26th Design*

Automation Conference, pp. 50-55, (1989).

[14] S. Olcoz and J. M. Colom, "Towards a formal semantics of VHDL IEEE Std. 1076-1987." *EURODAC with EUROVHDL 93*, Hamburg, September 1993.

[15] S. Olcoz, J. M. Colom, "Analysis Tools Applied to VHDL." *EuroMICRO 93*, Barcelona, September 1993.

[16] R. Piloty, M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill and P. S. Kelly, "CONLAN Report", *Lecture Notes in Computer Science*, no. 151, Springer-Verlag, 1983.