A VHDL Error Simulator for Functional Test Generation

Alessandro Fin   Franco Fummi
DST Informatica
Università di Verona, 37134 Verona, ITALY

Abstract

This paper describes an efficient error simulator able to analyze functional VHDL descriptions. The proposed simulation environment can be based on commercial VHDL simulators. All components of the simulation environment are automatically built starting from the VHDL specification of the description under test. The effectiveness of the simulator has been measured by using a random functional test generator. Functional test patterns produce, on some benchmarks, a higher gate-level fault coverage than the fault coverage achieved by a very efficient gate-level test pattern generator. Moreover, functional test generation requires a fraction of the time necessary to generate test at the gate level. This is due to the possibility of effectively exploring the test patterns space since error simulation is directly performed at the VHDL level.

1 Introduction

Design verification is becoming the most complex task of the entire design flow due to the continuous increasing of the design complexity. Formal verification techniques [1, 2, 3] would be the panacea if they would be widely applicable. However, their computation complexity is still prohibitive, thus the common practice for functional verification is still based on simulation. Some simulation-based verification approaches have been presented in the literature [4, 5, 6]. They usually compare the specification and the implementation of a device described by means of a hardware description language (HDL), such as VHDL [7] or Verilog [8]. Such techniques differ in the way the functional test patterns are identified since test patterns application to both specification and implementation and results comparison can be performed by using standard HDL simulators.

Whenever production testing is approached, test patterns are computed by means of defect and fault models related to structural descriptions of the device. However, the use of functional patterns for this task would be of extreme interest if they would be able to cover faults. In fact, functional patterns are used to validate all transformations trough the synthesis process and it would be a waste to discard them during production testing. Some approaches for functional testing have been investigated in the past years starting from microprocessors [10, 11] and finite state machine descriptions [12] until general specifications based on a HDL [13, 5, 14, 15]. These last approaches inject errors into a HDL descriptions and try to differentiate the erroneous behavior from the error-free behavior. An important component of such methodologies is the error simulator, which simplifies the error list by removing covered errors. Furthermore, if the functional test generation methodology is simulation based (e.g, based on random test generation or genetic algorithms [16]), the error simulator becomes the most crucial component due to the required high number of simulation sessions.

The aim of this paper is the presentation of a general error simulator for VHDL descriptions, which allows to use the most common metrics for functional testing, that is, statements coverage, branch coverage, condition coverage and path coverage [17]. The error simulator can be based on any commercial VHDL simulator, or, for improving performance, on the Model Technology simulator, which allows the mixed simulation of VHDL and C modules. The current implementation exploits this characteristic of the Model Technology simulator to generate a very efficient error simulator.

A fault model and a fault simulator for VHDL descriptions have been presented in [18]. This methodology produces a modified VHDL description for each fault and it proposes to simulate them with a standard VHDL simulator and to compare simulator output files to identify tested faults. No multiple faults can be handled, and VHDL must be recompiled at any new injected fault, but, more important, no simulation optimizations can be implemented, for instance no fault dropping can be performed. The simulation methodology presented in [19] proposes to dynamically inject faults in the VHDL code by controlling some extra inputs and it shows the general feasibility of VHDL fault simulation.

The proposed VHDL error simulator has the following characteristics:

- It analyzes any functional VHDL description.

- A modified version of the VHDL description is automatically generated based on a error list.

- Each error, or a set of errors, can be injected, at any time of the simulation process, in the description by driving control lines.

- Some synchronization modules are automatically generated, eventually in C language, to produce a closed design-entity, which performs all steps of the error simulation process: error injection, test application, response analysis and error list maintenance.

- The generated VHDL design-entity must be compiled once and can be simulated by a commercial VHDL simulator to perform error simulation.

The efficiency of the proposed simulator is measured in this paper by randomly generating functional test patterns. The possibility of exploiting error injection and simulation at the VHDL level allows random generation to produce very effective test patterns, and this is not usually true whenever performed at the gate level. The application of such patterns on the gate-level implementations of the functional VHDL specifications produces equal or even better fault coverage than the application of a commercial gate-level test pattern generator.

The paper is organized as follows. The proposed VHDL error simulation methodology is described in Section 2 with the description of the adopted VHDL error model and the error injection strategy. The simulator architecture is presented in Section 3, where some optimization techniques for error injection are also described. Section 4 presents some experimental results aiming at showing the effectiveness of the proposed VHDL error simulator. A random generator approach has been used and the achieved gate-level fault coverage is compared to the fault coverage achieved by an efficient commercial TPG working at the gate level.

2 Simulation Methodology

The proposed simulation methodology for VHDL is composed of the following elements (Figure 1):

- **VHDL Component to be Tested**. The design entity of the `VHDL component to be tested` is compiled in the LEDA database [20] and analyzed and modified through the database routines.

- **Error List**. The `Error List` is generated by the `Error List Generator` based on the error model described in [15] and briefly summarized in Section 2.1. It contains the list of all stuck-at bit errors to be injected. Each error is described by the type (stuck-at zero/one or stuck-at false/true), the target signal or variable and the bit position (see Figure 3).

- **Simulator Generator**. It is composed of a set of routines working on the LEDA database. By manipulating the `Error List` and the description of the `VHDL Component to be Tested`, such routines generate the `VHDL Component with Injected Errors` and the C descriptions of the `Error Driver`, the `Test Vector Generator` and the `Result Analyzer` (see Section 3).

- **Simulator Architecture**. The set of VHDL and C modules previously introduced can be executed in a VHDL simulation environment to efficiently measure the error coverage of the `VHDL Component to be Tested`.

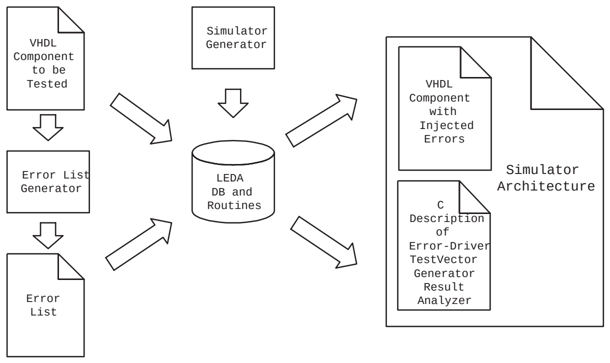
VHDL Component to be Tested
Simulator Generator
Error List Generator
LEDA DB and Routines
Error List
VHDL Component with Injected Errors
C Description of Error-Driver TestVector Generator Result Analyzer
Simulator Architecture

Figure 1. Simulator Developing Phase

2.1 VHDL Error Model

The adopted error model considers those failure modes of VHDL closely related to RT-level stuck-at faults [15]. It assumes a single error model composed of the following two kinds of errors:

Bit failures. Each variable, signal or port is considered as a vector of bits. Each bit can be stuck-at zero or one.

Condition failures. Each condition can be stuck-at true or stuck-at false, thus removing some execution paths in the erroneous representation.

The error model excludes explicitly the incorrect behavior of the elementary operators (e.g., +,-,*,...). Only single bit input or output errors are considered, therefore including all operator's equivalent errors.

It has been proven in [21] that this VHDL error model covers all statement, branch and condition errors, moreover it covers an important part of all path errors. For this reason, the simulation of the `Error List`, generated by using this error model, allows to represent the most common metrics for functional testing. However, the description of the potentialities of the choosen error model is not one of the goals of this paper, the interested reader is referred to [21].

2.2 Error Injection Strategy

The aim of the adopted error injection strategy is the simulation of erroneous behaviors at simulation time without requiring the re-compilation of the VHDL code or the modification of the simulator architecture. In this way, all operations concerning repeated injection of errors can be controlled by the simulator at run-time, thus allowing a functional test generator to efficiently perform many simulation cycles on different sets of errors. Control lines are added to the entity of the `VHDL Component to be Tested` to switch during simulation between the error-free and the erroneous behaviors, thus creating the entity of the `VHDL Component with Injected Errors`. Each error is controlled by using 2 bits: ’11’ represents a stuck-at one or stuck-at true, ’10’ a stuck-at zero or stuck-at false and ’0*’ represents no error. An example of `VHDL Component with Injected Errors` is shown in Figure 2.

```
ENTITY gcd32 IS
PORT (
              clock : IN bit;
              reset : IN bit;
              xi,yi : IN UNSIGNED (SIZE-1 DOWNTO 0);
              out : OUT UNSIGNED (SIZE-1 DOWNTO 0)
);
ARCHITECTURE behavioral OF gcd32 IS
BEGIN
   PROCESS
      VARIABLE x,y : UNSIGNED (SIZE-1 DOWNTO 0) ;
      VARIABLE temp : UNSIGNED (SIZE-1 DOWNTO 0) ;
   BEGIN
      WAIT UNTIL clock = ’1’;
      x := xi;
      y := yi;
      WHILE (x > 0) LOOP
          IF(x < y) THEN
            temp := y;
            y := x;
            . . .
```

Figure 2. A VHDL Component to be Tested

```
0> Error on yi row 18 occurrence 1
   List of bit-errors:
            **SA 1 on bit 31 ... 0     **SA 0 on bit 31 ... 0
1> Error on x row 25 occurrence 2
   List of bit-errors:
            **SA 1 on bit 31 ... 0     **SA 0 on bit 31 ... 0
```

Figure 3. Example of Error List

Let us consider, for instance, the first error reported in Figure 3. It represents the stuck-at one and zero error on all 32 bits of the `yi` signal. This error is controlled by the added input signal `y18er`, which is able to model all 64 stuck-at bit errors. The `y18_inj LOOP` checks all bits of such a signal and modifies the value of the `y` signal according to the errors to be injected. Thus, by controlling values of input signal `y18er` it is possible to simulate all 64 erroneous behaviors related to single errors or any combination of multiple errors.

Note that, the complexity of the VHDL code of the `component with injected errors` increases lineary in the number of inputs, signals and variables involved.

3 Simulator Architecture

```
ENTITY gcd32 IS
PORT (
              clock : IN bit;
              reset : IN bit;
              xi,yi, out : IN UNSIGNED (SIZE-1 DOWNTO 0);
              out : OUT UNSIGNED (SIZE-1 DOWNTO 0);

              y18err, x25err : IN ERROR_LINE;
);
ARCHITECTURE behavioral OF gcd32 IS
BEGIN
   PROCESS
      VARIABLE x : UNSIGNED (SIZE-1 DOWNTO 0) ;
      VARIABLE y : UNSIGNED (SIZE-1 DOWNTO 0) ;
      VARIABLE temp : UNSIGNED (SIZE-1 DOWNTO 0) ;
      VARIABLE op : UNSIGNED (SIZE-1 DOWNTO 0) ;
   BEGIN
      WAIT UNTIL clock = ’1’;
      x := xin;

      y := yin;
      y18_inj: FOR i IN 0 SIZE-1
      LOOP
         CASE y18err(i) IS
            WHEN STUCK0 => y(i) := ’0’;
            WHEN STUCK1 => y(i) := ’1’;
            WHEN OTHERS => null;
         END CASE;
      END LOOP y18_inj;

      WHILE (x > 0) LOOP
         IF (x < y) THEN
            temp := y;
            y := x;
            x := temp;
         END IF;

         op := x;
         x25_inj: FOR i IN 0 SIZE-1
         LOOP
            CASE x25err(i) IS
               WHEN STUCK0 => op(i) := ’0’;
               WHEN STUCK1 => op(i) := ’1’;
               WHEN OTHERS => null;
            END CASE;
         END LOOP x25_inj;
```

Figure 4. VHDL Component with Injected Errors

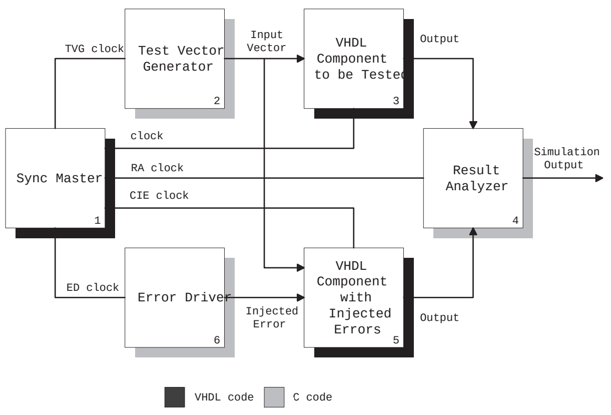

Figure 5. Simulator Architecture

Figure 5 shows the architecture of the proposed VHDL error simulator. It is composed of six modules automatically generated by the routines working on the LEDA database, as shown in Figure 1. Such routines produce all simulator elements by reading from the database the `VHDL Component to be Tested` and the `Error List`. In the current implementation some modules are written in VHDL and other modules in C to improve the simulation performances. However, if the used VHDL simulation environment does not allow such a mixed simulation, C modules can be rewritten in VHDL. Let us describe each module:

- **Sync Master**. This module is responsible for the synchronization of the other modules. It is connected to a different clock line for any module. It drives the `Test Vector Generator`, which generates new input vectors for both the `VHDL Component to be Tested` and the `VHDL Component with Injected Errors`. Moreover, the `Sync Master` commands the `Error Driver` to inject new errors in the `VHDL Component with Injected Errors` and commands the `Result Analyzer` to compare the output of both VHDL components, when they finished the computation.

- **Test Vector Generator**. This module produces test patterns for the `VHDL Component to be Tested` and the `VHDL Component with Injected Errors`. In the current implementation this module produces random patterns, however it could be replaced by any functional test generator since it is written in C and it can be interfaced to any other programs by using, for instance, the socket library. Note that the rest of the simulator architecture is left unchanged if the `Test Vector Generator` is modified.

- **Error Driver**. This module injects a single bit stuck-at error in the `VHDL Components with Injected Errors`, by driving the error control lines, as described in Section 2.2. The module manages an error table, which has been generated by the simulator generator starting from the `Error List`. The error table allows to identify detected errors and the corresponding detecting test patterns. Moreover, in fault dropping mode, only undetected errors are injected, thus sensibly improving the simulation efficiency.

- **Result Analyzer**. This module verifies if the current test pattern, generated by the `Test Vector Generator`, is able to detect the injected error. The analysis compares the outputs of the `VHDL Component to be Tested` and the outputs of the `VHDL Component with Injected Errors`. This comparison is performed every time the two components assert their respective `done` signals. Such signals have been inserted in the original VHDL description at the time an output port is assigned. The use of such `done` signals allows the synchronization of the error-free and erroneous VHDL components which could produce results at different times due to the injection of errors. The result of this component is the error coverage achieved at the end of the simulation.

- **VHDL Component with Injected Errors**. This VHDL component has been derived from the original VHDL description by adding error control signals and `done` signals as described before. If no error is injected the output of this component is equal to the output produced by the `VHDL Component to be Tested`.

4 Experimental Results

The proposed error simulation methodology has been implemented by using:

- The LEDA LVS Libraries [20] for VHDL parsing, error generation, error injection and simulator architecture generation. It is composed of about 10K C code lines and it has been inserted into the Commit [21] environment. This part automatically realizes the simulator architecture reported in Figure 5 starting from any VHDL design entity.

- The Model Technology VHDL simulator, which allows the linking and concurrent simulation of VHDL and C code. The interface between VHDL and C code is based on the FLI library, which converts C function parameters into VHDL input/output ports and vice versa.

All experiments have been performed on a SUN Ultra5 333MHz with 256MByte RAM.

The effectiveness of the proposed approach has been evaluated on some behavioral VHDL designs of the high-level-synthesis set [22]. Such benchmarks have been analyzed for different data size: 8, 16 and 32 bits to evaluate the dependency of the random functional test generator from the data size. Characteristics of the analyzed benchmarks are summarized in the leftmost part of Table 1 in terms of inputs, outputs and modeled VHDL errors. Two different combinations of scheduling and allocation have been performed on each benchmark by using the high-level synthesis tool Visual Architect [23]. Different clock cycle lengths produce such different combinations of scheduling and allocation. RTL and logic synthesis have been executed by using a commercial RTL synthesis tool. The rightmost part of Table 1 shows structural information for each benchmark in terms of scheduled control steps, number of gates, memory elements and stuck-at faults. Table 2 shows the average CPU run times necessary to simulate 1000 test vectors on

Table 2. Average simulation times.

Name	CPUs.	CPUs. `WEI`	$\Delta\%$
diffeq	542	595	9.8
ellipf	664	718	8.1
fir	712	786	10.4
gcd	323	365	13.0
average	487	544	10.3

Table 1. Characteristics of the analyzed benchmarks.

Name	BUS-size	In.bits	Out.bits	VHDL Errors	Ctrl.Steps	Gates	F.F.	Stuck-at Faults
diffeq	8	42	24	672	4	1679	83	5078
					7	2077	99	5160
	16	82	48	1344	4	4637	163	11448
					7	4988	195	12358
	32	162	96	2688	4	14026	323	30808
					7	14727	387	32720
ellipf	8	66	64	1424	5	1848	123	6432
					12	2300	140	5866
	16	130	128	2848	5	3900	243	12956
					12	4373	276	10926
	32	258	256	5696	5	8985	483	26708
					12	10540	548	22616
fir	8	194	8	1532	5	1507	43	4438
					12	2233	118	6482
	16	384	16	3056	5	2957	75	9258
					12	4570	262	14704
	32	768	32	6104	5	8329	139	21868
					12	8827	204	23672
gcd	8	18	8	164	3	398	26	1102
					7	636	35	1588
	16	34	16	324	3	781	50	2274
					7	1143	67	3338
	32	66	32	644	3	1649	98	4714
					7	2153	131	6386

the benchmarks before and after transformations for errors injection. The 32-bit versions of the benchmarks have been used. The average cpu time overhead produced by the transformation is only 10.3%.

The proposed VHDL error simulator has been used in conjunction with a random test generator. Test vectors are randomly generated as bit vectors. The size of these vectors is twice the sum of the size of the added signals for error injection (see the stuck-at error encoding presented in Section 2). The fault coverage of such random functional test patterns has been compared to the fault coverage achieved by running one of the most efficient commercial gate-level TPG, which performs at first random generation followed by deterministic test generation. Results of this comparison are reported in Table 3, where all versions of each benchmark are identified by a suffix composed of the data size and the number of control steps: e.g., the first benchmark of Table 1 is referred to as `diffeq_8_4`.

Table 3 shows the number of stuck-at faults (*Tot.F.*), the CPU time, the number of detected faults and the corresponding fault coverage for the random generation based on the proposed VHDL error simulator and for the deterministic gate-level TPG. For this second generator, the fault coverage randomly achieved at the gate level is also reported (*%Rnd.F.C.*). The gate-level fault coverage achieved by the VHDL random test generator is on average higher than the global fault coverage achieved by the gate-level TPG. Gate-level random fault coverage is a fraction of the functional random fault coverage. Moreover, functional test generation time is sensibly lower than gate-level generation time. These results highlight that the adoption of an efficient VHDL error simulator allows a functional test generator (even random) to effectively explore the test patterns space. To improve the fault coverage it would be necessary to adopt some deterministic techniques applicable to behavioural component descriptions [15].

5 Concluding Remarks

The paper has presented the architecture of an error simulator able to analyze functional VHDL descriptions. The simulation environment is based on a commercial VHDL simulator, which analyzes some modules automatically derived from the VHDL description to be tested. Errors are injected during simulation without being constrained to recompile any VHDL description. Injected errors are based on a error model, which guarantees to be equivalent to the well known metrics based on branch, condition and statement coverage. The proposed error simulator, in conjunction with a simple random test generator, produced test patterns achieving very high gate-level fault coverage in a fraction of the time required by an efficient gate-level TPG.

Future work will concern the use of the error simulator with more efficient functional test pattern generators based on implicit techniques and genetic algorithms and the extension of the simulator to sequential circuits and multiproces descriptions.

Table 3. VHDL random TPG versus gate-level deterministic TPG.

		Functional VHDL Random TPG			Gate-level Deterministic TPG			
Name	Tot.F.	CPUs.	Det.F.	%F.C.	CPUs.	%Rnd.F.C.	Det.F.	%F.C.
diffeq_8_4	4896	9	4524	**92.4**	221	0.0	4269	87.2
diffeq_8_7	5156	9	4723	**91.6**	220	0.0	4676	90.7
diffeq_16_4	11068	98	10334	**93.4**	2382	0.0	9607	86.8
diffeq_16_7	11916	98	11030	92.6	1282	3.6	1124	**94.2**
diffeq_32_4	30044	595	28064	**93.4**	19388	3.9	27100	90.2
diffeq_32_7	31830	595	29520	92.7	6097	3.6	30524	**95.9**
ellipf_8_5	6190	62	5564	89.9	171	13.0	5571	**90.0**
ellipf_8_12	5830	62	5427	93.1	113	8.9	5451	**93.5**
ellipf_16_5	12114	211	11316	**93.4**	750	9.6	10951	90.4
ellipf_16_12	10828	211	10150	93.7	290	8.8	10146	93.7
ellipf_32_5	25746	718	23537	**91.1**	4186	10.3	23377	90.8
ellipf_32_12	22380	718	21054	94.1	1120	9.3	2160	94.1
fir_8_5	4182	89	3855	92.2	257	11.2	3889	**93.0**
fir_8_12	6218	89	5733	**92.2**	793	8.2	5024	80.8
fir_16_5	8994	312	8427	93.7	760	7.6	8472	**94.2**
fir_16_12	12868	312	11954	**92.9**	8637	9.8	8583	66.7
fir_32_5	20034	786	18050	90.1	3667	11.1	19313	**96.4**
fir_32_12	21496	786	19625	91.3	5616	9.3	20700	**96.3**
gcd_8_3	1094	3	1041	**95.0**	18	6.6	1025	93.7
gcd_8_7	1575	3	1500	**95.2**	290	4.6	1462	92.8
gcd_16_3	2248	72	2112	**94.0**	193	6.0	2043	90.9
gcd_16_7	3260	72	3079	**94.4**	890	4.1	2090	64.1
gcd_32_3	4637	365	4293	**92.6**	2184	5.6	3450	74.4
gcd_32_7	6234	365	5809	**93.2**	4020	4.2	4208	67.5
average				**92.8**		6.7		88.8

Acknowledgments

We would like to acknowledge Prof.Sciuto and Dot.Ferrandi of Politecnico di Milano for the useful discussions on functional testing.

References

[1] T. Filkorn, M. Payer, and P. Warketin. Symbolic verification of high-level synthesis results from Callas. *Proc. 6th International Workshop on High-Level Synthesis*, pages 344–353, nov 1992.

[2] O. Coudert, C. Berthet, J.C. Madre. Verification of Sequential Machines Based on Symbolic Execution. *Proc. Workshop on Automatic Verification Methods for Finite State Systems*, 1989.

[3] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, D.L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Trans. on CAD/ICAS*, 13(4):401–424, April 1994.

[4] R. Vemuri and R Kalyanaraman. Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint programming. *IEEE Trans. on VLSI Systems*, 3(2):201–214, June 1995.

[5] F. Fallah, P. Ashar, and S. Devadas. Simulation vector generation from HDL descriptions for observability-enhanced statement coverage. *Proc. ACM/IEEE DAC*, pages 666–671, 1999.

[6] F. Fallah, S. Devadas, and K. Keutzer. Functional vector generation for HDL models using linear programming and 3-satisfaiability. *Proc. ACM/IEEE DAC*, pages 528–533, 1998.

[7] IEEE standard VHDL language reference manual. *IEEE Sid 1076-1993, The Institute of Electrical and Electronic Engineerings, Inc., New York, NY*, 1993.

[8] D.E. Thomas, P. Moorby. The Verilog Hardware Description Language. *Kluwer Academic Publisher, Nowell Massachusetts*, 1991.

[9] Synopsys VSS user's manual. *Synopsys*, 1998.

[10] D. Brahme and J.A. Abraham. Functional testing of microprocessors. *IEEE Trans. on Computers*, C-33(6):475–485, 1985.

[11] J. Lee and J.H. Patel. Architectural level test generation for microprocessors. *IEEE Trans. on CAD/ICAS*, 13(10):1288–1300, October 1994.

[12] K.T. Cheng and A.S. Krishnakumar. Automatic generation of functional vectors using the extended finite state machine model. *ACM Trans. on design Automation of Electronic Systems*, 1(1):57–59, January 1996.

[13] U.H. Levendel and P.R. Menon. Test generation algorithms for computer hardware description languages. *IEEE Trans. on Computers*, C-31(7):557–588, July 1982.

[14] S.R. Rao, B.Y. Pan, and J.R. Armstrong. Hierarchical test generation for VHDL behavioral models. *Proc. European Design Automation Conference*, pages 175–182, 1993.

[15] F. Ferrandi, F. Fummi, and D. Sciuto. Implicit test generation for behavioral VHDL models. *Proc. IEEE ITC*, pages 436–441, 1998.

[16] E.M. Rudnick, R. Vietti, A. Ellis, F. Corno, P. Prinetto and M. Sonza Reorda. Fast Sequential Circuit test Generation Using High-Level and Gate-Level Techniques. *Proc. IEEE DATE*, pages 570–576, 1998.

[17] Glenford J. Myers. *The Art of Software Testing*. Wiley - Interscience, New York, 1979.

[18] T. Riesgo, J. Uceda. A Fault Model for VHDL Descriptions at the Register Transfer Level. *Proc. EURO-DAC/EURO-VHDL*, 1996.

[19] F. Celeiro, L. Dias, J. Ferreira, M.B. Santos, J.P. Texeira. VHDL Fault Simulation for Defect-Oriented Test and Diagnosis of Digital ICs. *Proc. EURO-DAC/EURO-VHDL*, 1996.

[20] LEDA VHDL*Verilog System user's manual. *VHDL Compiler Version 4.1*, 1993.

[21] F. Ferrandi, F. Fummi, and D. Sciuto. Design Verification of VHDL Specifications through Functional Testing. *Internal Report 3-99*, Universitá di Verona, 1999.

[22] 1991 and 1992 high level synthesis benchmarks. *ftp://mcnc.mcnc.org/pub/benchmark/HLSynth91[92]*, 1992.

[23] Visual architect user's manual. *Cadence*, 1998.