Effective Techniques for High-Level ATPG

Fulvio Corno, Gianluca Cumani, Matteo Sonza Reorda, Giovanni Squillero

Politecnico di Torino
Dipartimento di Automatica e Informatica
http://www.cad.polito.it/

Abstract

The ASIC design flow is rapidly moving towards higher description levels, and most design activities are now performed at the RT-level. However, test-related activities are lacking behind this trend, mainly since effective fault models and test pattern generation tools are still missing. This paper proposes techniques for implementing a high-level ATPG. The proposed algorithm mixes a code coverage-oriented approach with fault-oriented optimizations. Moreover, it exploits a fault model at the RT-level that enables efficient fault simulation and guarantees good correlation with gate-level fault coverage. Experimental results show that the achieved results are comparable or better than those obtained at the gate level or by similar RT-level approaches.

1. Introduction

In recent years the application-specific integrated circuit (ASIC) design flow experienced radical changes. Deep sub-micron integrated circuit (IC) manufacturing technology is enabling designers to put millions of transistors on a single integrated circuit. Following Moore's law, design complexity is roughly doubling every 12-18 months. In addition, there is an ever-increasing demand on reducing time to market. With complexity skyrocketing and such a competitive pressure, designing at high levels of abstraction has become more of a necessity than an option.

In this scenario, high-level test pattern generation is increasing its industrial relevance [15]. Designers would like to foresee an ASIC testability before starting its logic synthesis. The design practice is pushing the insertion of design for testability (DfT) structures up to the register-transfer (RT) level, and their effectiveness should be evaluated as soon as possible. In addition, it has been increasingly observed that gate-level sequential automatic test pattern generation (ATPG) techniques may take unacceptable amounts of computing time and resources to tackle larger sequential circuits unless design-for-testability structures are used. High-level ATPG tools are expected to exploit compact information about design structure and behavior, and to generate high-quality test sequences more efficiently. Moreover, it is supposed that high-level generated test benches could be able to detect faults that would be very hard for gate-level ATPGs [18].

This paper presents Prince, an algorithm for implementing a high-level ATPG. The proposed technique mixes a code coverage-oriented approach with fault-oriented optimizations.

Section 2 sketches some background information, Section 3 details the algorithm, Section 4 reports some experimental results and Section 5 concludes the paper.

2. Background

Tackling test issues above the gate level is a hard task, and the lack of a fault model is one of the hardest theoretical barriers.

Code-coverage based fault models, deriving from the software testing field, may seem suitable to be applied on HDL descriptions. However, coverage metrics such as line/block coverage, branch/conditional coverage, expression coverage and path coverage lack of direct relationships with gate-level stuck-at faults, and their applicability in the field of test is difficult. Other considerable difficulties stem from the large amount of concurrency, from the complexity of timing schemes and from the combined presence of behavioral and structural description styles. But, definitely, the main problem with code-coverage based fault models is probably the lack of an explicit observability concept. Coverage metrics only consider *reachability*, that is like *fault controllability* in the gate-level domain. However, any ATPG should tackle faulty-behavior *observation* as well [5].

[16] presents TAO, a two-pass approach using a symbolic RTL test generator. The proposed testing paradigm involves writing path equations for modules, given the RTL connectivity, and solving them to obtain regular expressions for control paths.

Probably, the most successful proposal of a hardware-related high-level fault model is *Observability-Enhanced Statement Coverage* [7]. It introduces the concept of *tag* as the possibility that an incorrect value is computed at a given location, thus approximating the effects of fault propagation. Since this fault model does not assume any specific fault effect, its generality prevents explicit fault simulation.

The first ATPG exploiting *Observability-Enhanced Statement Coverage* was presented in [6]. The vector generation procedure is based on hybrid linear programming and Boolean satisfiability methods.

ARTIST, a different RT-level ATPG exploiting high-level information to reach high code-coverage figures, was presented in [3]. Differently from [6], ARTIST is a simulation-based approach. It is based on an evolutionary algorithm coupled with a commercial VHDL simulator, and due to the adoption of a commercial tool, it is able to produce sequences for general synthesizable VHDL description, with few limitations in complexity and characteristics, and it does not require any effort for re-modeling circuits or extracting special information. However, neglecting observability, sequences generated by ARTIST are not optimized for test purpose.

In [4], ARTIST code-coverage metric was augmented with simplified observability. Fault-coverage figures dramatically increased, but the lack of a real fault model prevented the usage of a fault-dropping mechanism. ARTIST was given the goal to increase an observability measure, without meaningful stopping condition. Thus, the approach was not suitable for larger designs.

In [1] an extension of observability-enhanced statement coverage was proposed. In the new model, explicit *RT-level assignment single-bit* stuck-at's are used instead of generic tags. An RT-level assignment single-bit stuck-at fault is defined as a single-bit stuck-at in the effect of an RT-level assignment operation: when a fault is present, the affected object (signal or variable target of an assignment statement) loads the correct value, except for one bit that is forced to 0 or 1. Experimental figures show that this model is highly correlated with gate-level coverage.

In [8] Ferrara et al. presented BEHATE, an RT-level tool based on a metric called *bit-coverage*, close to the *RT-level assignment single-bit*. Although the paper is aimed at functional verification, experimental results show a strong relation between high- and gate-level faults.

[2] shows a simulation techniques based on simulation command scripts that allows efficient exploitation of *RT-level assignment single-bit* faults. Using the Tcl interface of a commercial simulator, the simulation of each faulty circuit is shown no more costly than simulation of the original circuit.

This paper presents new techniques for devising a high-level ATPG process. The proposed algorithm, described in the next sections, mixes a coverage-oriented approach with fault-oriented optimizations. First, the RT-level circuit description is automatically analyzed to extract static structural information, control and data dependencies, and to group statements in basic-blocks. Then a code coverage approach is exploited to excite the RT-level assignment single-bit faults. After excitation, fault effect propagation and observation are tackled utilizing simulation scripts. In conclusion, a fault dropping phase is run to optimize the process.

3. ATPG System

Prince, the ATPG algorithm (Figure 1), was developed exploiting the RT-level assignment single-bit fault model [1] and a simulation-based approach [2]. It incrementally builds a test set, adding new sequences through different stages, mixing a coverage-oriented approach with fault-oriented optimizations.

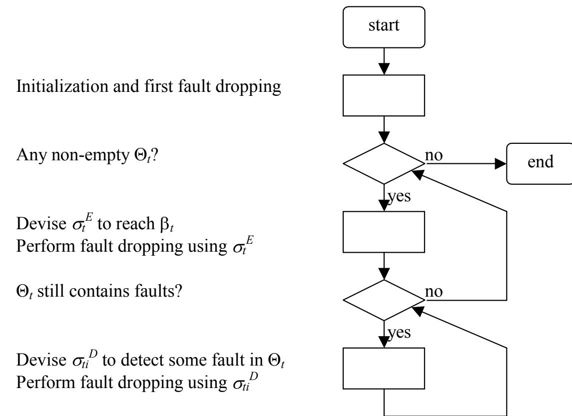

Figure 1: *Overall algorithm*

In different steps Prince exploits a genetic algorithm (GA) to seek new sequences. The goals of the different stages are different, thus the GA is given different fitness functions and different initial conditions. However, its general structure is the same.

The GA in Prince evolves a population of p individuals with an offspring ratio of p_o in each generation. It implements a steady-state evolution, i.e., new individuals are first added to the population, and then the p fittest ones are chosen for survival. Individuals are selected for reproduction using their linearized fitness. With a probability p, the new individual is built mutat-

ing a single parent: the original sequence can be shortened, or enlarged, or some bits may be flipped. Otherwise the new individual is built mating two sequences: it can inherit the beginning from one parent and the end from the other, or some entire bit column from each parent. The GA evolves until the goal is reached, or until the maximum number m_g of generations have been evaluated, or after m_s generations without any fitness improvement in the best individual.

At the end of the process, sequences may easily be compacted with a simple algorithm. Next Sections details the process.

3.1. Fault Model

The RT-level single-bit stuck-at fault model was presented in [1]. In this model, a fault is defined as a single-bit stuck-at in the effect of an RT-level assignment operation: when a fault is present, the affected object (signal or variable target of an assignment statement) loads the correct value, except for one bit that remains stuck to 0 or 1.

Faults are *single* and *permanent*: only one fault is inserted at a time and the fault effect is present during the whole simulation. The RT-Level single-bit stuck-at fault model does not explicitly consider control-flow faults, such as *stuck-at-true* or *stuck-at-false*.

Initially, the Fault List contains the list of all RT-Level single-bit stuck-at faults. However, during synthesis the RT level VHDL description is optimized in order to create an efficient gate-level design. The optimization process analyzes the VHDL description and simplifies all logic eliminating redundancies. In this phase some RT-level stuck-at faults lose their correspondent gate-level faults. In order to prevent this discrepancy is necessary to identify which parts of the logic described at the RT-level disappear during the optimization phase of the synthesis process and to eliminate the associated faults from the Fault List.

To perform Fault Simulation a serial fault simulation strategy is adopted. The good and each faulty machine are simulated, comparing their outputs. A fault is marked as *detected*, if it produces a difference on a Primary Outputs of the circuit at the end of a clock cycle. To run the simulations, the Test Pattern is first transformed to a set of commands that force the correct waveform for input signals, and the Fault List is transformed to a set of script commands for injecting faults during simulation.

Fault injection is made possible by creating routines that *change* the target signal/variable bit value during simulation, using the simulator scripting language (Tcl), when a given target assignment instruction is executed. The fault injection procedures must face various issues derived from the fault model, from VHDL Semantics and from the simulator itself.

Further details can be found in [2].

3.2. Analysis

The analysis aims at building a simplified internal model of the circuit.

Static structural information, control dependencies and data dependencies are extracted. The RT-level hierarchy is analyzed and processes broken down into basic blocks, i.e., blocks of statements that are guaranteed to be always executed together.

Then a correlation matrix C is inferred mixing the control-flow analysis with data dependencies. Let β_i and β_j be two basic blocks, the element c_{ij} of the correlation matrix C estimates the conditional probability that β_j will be executed given the execution of β_i.

The analysis is an automatic process performed through commercial tools for parsing VHDL. Each circuit has to be been analyzed only once, since information gathered during analysis does not depends on the results of the ATPG process.

This step is identical to the first step in [4].

3.3. Initialization

The initialization goal is to remove easy-to-detect faults.

Prince starts its GA to cultivate a sequence that maximizes the basic-block coverage. The initial population is generated randomly. The fitness function simply counts the number of covered basic blocks, without exploiting the knowledge of design structure. See function (1) in Figure 2.

Once such a sequence is generated, it is added to the final test set and simulated against all RT-level faults (see *Fault Dropping*, below).

$$\text{Fitness}_1(\sigma) = \sum_i \text{ExecutionCount}(\sigma, \beta_i) \quad (1)$$

$$\text{Fitness}_2(\sigma) = \sum_i \text{ExecutionCount}(\sigma, \beta_i) \cdot c_{it} \quad (2)$$

$$\text{Fitness}_3(\sigma) = \sum_{f \in \Theta_i} k_1 \cdot \text{Det}(f) + k_2 \cdot \text{Obs}(f) + k_3 \cdot \text{Exc}(f) \quad (3)$$

Figure 2: Different fitness functions

3.4. Basic-Block Fault Detection

After easy-to-detect RT-level faults have been removed, Prince starts the main ATPG process.

Let β_i be the i-th basic block, and Θ_i the set of all still undetected RT-level single-bit faults on assignments performed inside β_i. The basic-block fault detection stage is iterated for each non-empty Θ_i.

Let Θ_t be a non-empty set of faults selected as target for the fault detection stage. Prince first tries to create a sequence σ_t^E able to cover basic block β_t. The search exploits the same GA described in 3. The fitness function counts each activated basic block β_a weighting it with its correlation with the target β_t. See function (2) in Figure 2.

If such a sequence σ_t^E is found, it is guaranteed that all statements of β_t are *executed*, however it is not certain that faults in Θ_t are *excited*: the faulty value of the bit may be the same as the good one. Nevertheless, since σ_t^E is potentially a useful sequence and it is added to the final test set. When the GA is halted, the set Ψ of the p sequences in the last population is stored for later usage.

At this point, Prince start trying to *detect* all faults f_{ti} of Θ_t. Here, as in gate-level ATPG, "detect" implies *exciting* the fault and then *observing* it, by propagating its effect to a primary output. At the present, Prince exploits the simulation mechanism described in [2]: a loose interaction with a commercial VHDL simulator carried out through Tcl scripts.

The new step still exploits the GA described above. The initial population is loaded from Ψ and a third fitness function is adopted. This fitness function measures how many faults in Θ_t have been detected, how close is the sequence to observe new faults and how many faults in Θ_t have been excited. The three contributions are weighted in decreasing importance, thus observing is more important than exciting, and so on. See function (3) in Figure 2.

Once a sequence σ_{ti}^D able to detect new faults is found, it is added to the final test set. Then all faults detected by σ_{ti}^D are removed from Θ_t. It should be noted that removing detected faults from Θ_t leads to a change in the fitness function, because during fitness calculation only faults in Θ_t are considered. Finally, the whole population is re-evaluated and sorted according to the new criteria. Detection is iterated until Θ_t is empty or the GA aborts.

3.5. Fault Dropping

Each time a new sequence σ is added to the final test set, it is simulated against all still undetected RT-level faults. The fault-dropping phase exploits structural information. Prince simulates σ to get the list of covered basic blocks, and only the faults on covered statements are injected.

The speed-up archived by fault dropping is considerable. When Prince tackles the b14 benchmark, for instance, more than 7,500 RT-level faults (67% of the total) are dropped in this stage. However, it should be pointed out that it is the most time-consuming step of the algorithm. RT-level fault simulation, in general, is a resource-consuming task. In the proposed approach the circuit is not modified and does not need to be recompiled, however relying on an external commercial simulator introduces some overhead. Forcing new values during simulation, several RT-level faults cause overflows and boundary-check errors. Prince can handle all these exceptions, but it cannot handle them efficiently. When the simulator hangs, its process must be killed using unix signals and it must started again.

4. Experimental Results

To evaluate observability techniques, we analyze the ITC'99 VHDL RT-level benchmark circuits. Circuit characteristics are summarized in Table 1.

Circuit			VHDL			Gate-level		
	PI	**PO**	**#line**	**#p**	**Faults**	**#gate**	**#FF**	**faults**
b01	2	2	110	1	137	46	5	258
b02	1	1	70	1	80	28	4	150
b03	4	4	141	1	242	149	30	822
b04	11	8	102	1	635	597	66	3,356
b05	1	36	332	3	1,144	935	34	5,552
b06	2	6	128	1	207	60	9	302
b07	1	8	92	1	349	420	49	2,404
b08	9	4	89	1	153	167	21	918
b09	1	1	103	1	212	159	28	900
b10	11	6	167	1	258	189	17	1,054
b11	7	6	118	1	458	481	31	2,868
b12	5	6	569	4	1,272	1,036	121	6,084
b13	10	10	296	5	421	339	53	1,818
b14	32	54	509	1	11,054	4,775	245	28,990
b15	36	70	671	3	4,546	8,893	449	55,568
b20	32	22	1,085	3	20,882	9,419	490	57,794
b21	32	22	1,089	3	20,882	9,803	490	60,052
b22	32	22	1,613	4	33,462	15,071	735	92,536

Table 1: *Benchmark characteristics*

The first columns report the benchmark names, and the number of Primary Inputs and Outputs. The second column group reports data about RT-level descriptions: number of VHDL lines, number of VHDL processes (#p), and number of RT-level faults. Gate-level descriptions are also available and the following column group reports their characteristics in terms of number of combinational gates, number of Flip-Flops and number of stuck-at faults.

In the experiments, the population is composed of p=50 individuals, with an offspring ratio of p_o=60%.

Thus, in each generation 30 new sequences are first generated, then selection is performed on the whole set of 50+30 individuals. The mutation rate was set to 0.3, hence in 30% of the cases, the new individual is built mutating a single parent, while in 70% of the cases the new individual is built mating two different sequences. m_g was set to 50, and m_s was set to 10 for all circuits.

Experiments were run on a Sun Enterprise 250 running at 400 MHz and equipped with 2 Gbytes of RAM. CPU times required range from some hours to two days and are mainly due to the lack of flexibility of the commercial RT-level simulator. The efficiency of the approach would greatly increase whenever a closer interaction with the simulation core will be available.

Table 2 summarizes the results achieved by Prince on the benchmark circuits. The length of the test set (after compaction) is reported in the second column. Next column reports the fault coverage attained on RT-level fault list. After generation, test sets were simulated against gate-level netlists and the stuck-at fault coverage is reported in the last column.

Circuit	**Vec**	**Fault Coverage [%]**	
		RT	**Gate**
b01	44	95.62	100.00
b02	35	98.75	99.33
b03	139	75.62	74.82
b04	852	59.53	90.52
b05	138	51.14	33.43
b06	49	90.34	97.35
b07	89	71.92	58.28
b08	1,388	84.97	97.49
b09	1,102	92.45	85.33
b10	325	82.95	91.37
b11	2,121	78.82	91.77
b12	3,329	35.77	40.83
b13	4,193	76.96	84.76
b14	11,565	85.15	81.84
b15	864	44.57	23.75
b20	154,974	89.18	87.70
b21	51,463	89.31	88.21
b22	91,547	86.48	86.28

Table 2: *Prince Results*

In Table 3 gate-level stuck-at fault coverage are compared with previous works (the Fault Coverage represents the percentage of detected faults in the fault list). Table 3 compares the proposed approach with two RT-level tools: ARTIST (column *3*) and the observability-enhanced version of ARTIST (*4*). It would also have been interesting compare Prince with BEHATE [8], an RT-level tool developed by Ferrandi et al. However, authors reported the number of detected faults without mentioning which fault list they are using. And these numbers are higher than the number of both collapsed and un-collapsed faults reported in [3]. For the sake of comparison, Table 3 also reports data for a state-of-the-art commercial tool (*Comm.*) and a state-of-the-art academic tool (*12*).

Circ.	**RT-level**			**gate-level**	
	Prince	**[3]**	**[4]**	**[12]**	**Comm.**
b01	100.00	100.00	100.00	*n.a.*	100.00
b02	99.33	99.33	99.33	*n.a.*	99.33
b03	74.82	74.33	74.82	*n.a.*	74.82
b04	90.52	89.42	91.03	*n.a.*	91.51
b05	33.43	33.50	33.50	*n.a.*	33.38
b06	97.35	97.02	97.35	*n.a.*	97.35
b07	58.28	57.53	58.28	*n.a.*	57.28
b08	97.49	86.27	71.68	*n.a.*	98.15
b09	85.33	81.33	81.33	*n.a.*	90.56
b10	91.37	90.42	90.99	*n.a.*	92.22
b11	91.77	85.98	90.62	92.19	81.00
b12	40.83	45.99	44.49	49.03	21.17
b13	84.76	68.37	*n.a.*	*n.a.*	59.19
b14	81.84	79.65	*n.a.*	*n.a.*	95.04
b15	23.75	31.96	*n.a.*	*n.a.*	16.26
b20	87.70	79.99	*n.a.*	*n.a.*	26.57
b21	88.21	82.61	*n.a.*	82.81	55.14
b22	86.28	71.59	*n.a.*	*n.a.*	55.79

Table 3: *Stuck-at comparison*

Experimental results show that Prince is usually superior and at least comparable to both versions of ARTIST, the original one presented in [3] and the observability-enhanced one presented in [4]. It is remarkable that Prince was able to generate test sequences even for the larger benchmarks, while the observability-enhanced ARTIST cannot tackle circuits bigger than b12.

Compared to gate-level approaches, results are convincing. The attained Fault Coverage is higher than the commercial ATPG and, for b21, considerably higher than the state-of-the-art academic approach. Yet the few data presented in [12] prevent a more insightful comparison.

Benchmarks b12 and b15 are difficult even for gate-level ATPGs, but they deserve some comments. They both need extremely long and specific test sequences to activate all functionalities (b12 implements a *guess-a-sequence* game, b15 a microprocessor). For the sake of performance, Prince was pushed to avoid such a long sequences and this choice may have penalized it. In fact, even the statement coverage figures for the two benchmarks are quite low: 68.72% for b12 and 63.04%

for b15. More experiments are being performed to better understand this behavior.

5. Conclusions

Due to the wide adoption of logic synthesis tools, RT-level ATPG techniques are increasingly necessary in order to shift test-related activities towards the description level adopted by designers. A crucial point for developing effective high-level ATPGs lies in the identification of a suitable fault model, which should guarantee a good correlation with gate-level fault coverage figures while allowing the implementation of an ATPG algorithm.

This paper presented Prince, an algorithm for implementing a high-level ATPG, exploiting code coverage-oriented approach with fault-oriented optimizations. Prince adopts a fault model at the RT-level that enables efficient fault simulation and guarantees good correlation with gate-level fault coverage.

Experimental results showed that Prince is broadly applicable, and it attains fault coverage figures usually superior and at least comparable to other RT-level approaches. Also compared to gate-level approaches, results are considerable.

The two cases in which the approach is less satisfactory were analyzed and are currently under a deeper study.

6. Acknowledgments

The authors wish to thanks Fabio Salto for his help in implementing Prince and for performing all the experiments.

7. References

[1] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "An RT-level Fault Model with High Gate Level Correlation," *IEEE International High Level Design Validation and Test Workshop*, November 8-10, 2000

[2] F. Corno, G. Cumani, M. Sonza Reorda, Giovanni Squillero, "RT-level Fault Simulation Techniques based on Simulation Command Scripts," *DCIS 2000: XV Conference on Design of Circuits and Integrated Systems*, November 21-24, 2000

[3] F. Corno, M. Sonza Reorda, G. Squillero, "RT-Level ITC 99 Benchmarks and First ATPG Results," *IEEE Design & Test, Special issue on Benchmarking for Design and Test*, July-August 2000, pp. 44-53

[4] F. Corno, M. Sonza Reorda, G. Squillero, "High-Level Observability for Effective High-Level ATPG," *VTS2000: 18th IEEE VLSI Test Symposium*, May 2000, pp. 411-416

[5] S. Devadas, A. Ghosh, K. Keutzer, "An Observability-Based Code Coverage Metric for Functional Simulation," *Proceedings IEEE/ACM International Conference on Computer Aided Design*, 1996

[6] F. Fallah, P. Ashar, S. Devadas, "Simulation Vector Generation from HDL Descriptions for Observability-Enhanced Statement Coverage," *35th Design Automation Conference*, 1999, pp. 666-671

[7] F. Fallah, S. Devadas, K. Keutzer, "OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification," *34th Design Automation Conference*, 1998

[8] G. Ferrara, F. Ferrandi, A. Fin, F. Fummi, D.Sciuto, "Functional Test Generation for Behaviorally Sequential Models", *Proceedings IEEE Design Automation and Test in Europe Conference (DATE)*, Muenchen, Germany, 13-16 Marc 2001, pp.403-410.

[9] F. Ferrandi, F. Fummi, L. Gerli, D. Sciuto, "Symbolic Functional Vector Generation for VHDL Specifications," *35th Design Automation Conference*, 1999, pp. 442-446

[10] F. Ferrandi, G. Ferrara, D. Sciuto, A. Fin, F. Fummi, "Functional Test Generation for Behaviorally Sequential Models", Proc. IEEE Design Automation and Test in Europe Conference (DATE), Muenchen, Germany, 13-16 Marc 2001, pp.403-410

[11] A. Fin, F. Fummi, "A VHDL Error Simulator for Functional Test Generation," *IEEE European Design, Automation and Test Conference*, 2000, pp. 390-395

[12] A. Giani, S. Sheng, S. Hsiao, I. Agrawal, "Compaction-Based Test Generation Using State and Fault Information", in *Proceedings Asian Test Symposium*, pp. 159-164, 2000

[13] A. Giani, S. Sheng, M. Hsiao, V. Agrawak, "Efficent Spectral techniques for Sequential ATPG", Proceedings of the IEEE Design Automation and Test in Europe Conference, March, 2001

[14] H. S. Hsiao, E. M. Rudnick, J. H. Patel, "Dynamic State Traversal for Sequential Circuit Test Generation", ACM TODAES, vol. 5, n. 3, July 2000, pp. 548-565

[15] High Time for High-Level Test Generation*, Panel at the IEEE International Test Conference*, 1999, pp. 1112-1119

[16] S. Ravi, G. Lakshminarayana, and N. K. Jha, "TAO: Regular expression based high-level testability analysis and optimization", in *Proceedings International Test Conference*, pp. 331–340, 1998

[17] P. A. Thaker, V. Agrawak, M. Zaghloul, "Register-Transfer Level Fault Modeling and Test Evaluation Techniques for VLSI Circuits", *Proceedings International Test Conference (ITC2000)*, 2000, pp. 940-949

[18] M. B. Santos, F. M. Gonçalves, I.C. Texeira, J. P. Texeira, "RTL-Based Functional Test Generation for High Defects Coverage in Digital SOCs", *IEEE European Test Workshop*, 2000, pp. 99-104