HIGH-LEVEL AND HIERARCHICAL TEST SEQUENCE GENERATION

Gert Jervan, Zebo Peng

Linköping University
Embedded Systems Laboratory
Linköping, Sweden

Olga Goloubeva, Matteo Sonza Reorda, Massimo Violante

Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy
www.cad.polito.it

Abstract

Test generation at the gate-level produces high-quality tests but is computationally expensive in the case of large systems. Recently, several research efforts have investigated the possibility of devising test generation methods and tools to work on high-level descriptions. The goal of these methods is to provide the designers with testability information and test sequences in the early design stages. The cost for generating test sequences in the high abstraction levels is often lower than that for generating test sequences at the gate-level, with comparable or even higher fault coverage. This paper first analyses several high-level fault models in order to select the most suitable one for estimating the testability of circuits by reasoning on their behavioral descriptions and for guiding the test generation process at the behavioral level. We assess then the effectiveness of high-level test generation with a simple ATPG algorithm, and present a novel high-level hierarchical test generation approach to improve the results obtained by a pure high-level test generator.

Introduction

In the last years, new techniques have been developed to integrate an entire system on a single chip, called System-on-Chip (SOC). SOC products represent a real challenge not only from the manufacturing point of view, but also when design issues are concerned.

To cope with the challenges faced by SOC designers, tools and techniques dealing with design at higher levels of abstraction are becoming an industrial reality. In particular, behavioral-level synthesis tools and the more recently introduced co-design environments are starting to play an important role in the initial phases of the design process. The major benefit stemming from these design environments is the possibility of quickly evaluating the costs and benefits of different architecture alternatives, including both hardware and software components, starting from the algorithms a SOC should implement.

While the design practice is quickly moving toward higher levels of abstraction, test issues are still considered only when a detailed description of the design is available, typically at the gate level for test sequence generation and at register transfer (RT) level for design for testability structure insertion.

Recently intensive research efforts have been devoted to devise solutions tackling test sequence generation in the early design phases, mainly the RT level, and several approaches have been proposed [14]. Most of them are able to generate test patterns of good quality, sometimes comparable or even better than those of gate-level ATPG tools. However, lacking general applicability, these approaches are still not accepted by the industry. The different approaches are based on different assumptions and on a wide spectrum of distinct algorithmic techniques. Some are based on extracting from a behavioral description the corresponding control machine [1] or the symbolic representation based on binary decision diagrams [2], while others also synthesize a structural description of the data path [3]. Some approaches rely on a direct examination of the HDL description [4], or exploit the knowledge of the gate-level implementation [5]. Some others combine static analysis with simulation [6].

Most of the cited approaches rely on high-level fault models for behavioral HDL descriptions that have been developed by the current practice of software testing [7], and extend them to cope with hardware descriptions. Several authors have proposed alternative fault models. Nevertheless, a reference fault model playing, at the behavioral level, the same role the well-known stuck-at one is playing at the gate level, is still missing.

This paper first analyzes several high-level fault models in order to select the most suitable one for estimating the testability of circuits by reasoning on their behavioral descriptions and for guiding the test generation process at the behavioral level. We assess then the effectiveness of high-level test generation process with a simple ATPG algorithm, and present a novel high-level hierarchical test generation approach for improving the results obtained by a pure high-level test generator. The hierarchical test generator takes into account structural information from lower levels of abstraction while generating test sequences on the behavioral level.

High-Level Fault Models

When test issues are addressed at an abstraction level higher than the traditional gate-level the first problem that

must be addressed is the identification of a suitable high-level fault model. By working on system models that neglect the detailed information gate-level netlists have, the high-level fault models are not able to precisely foresee the gate-level fault coverage, which is normally used as the reference measure to quantify a circuit's testability. Nevertheless, they can be exploited to rank test sequences according to their testability value. The most common high-level fault models proposed in literature as metrics of the goodness of test sequences when working at higher levels of abstraction (RT level and behavioral level) include the following:

- **Statement coverage:** this is a well-known metric in the software testing field [7] indented to measure the percentage of statements composing a model that are activated by a given test sequence. Further improvements of this metric are the *Branch coverage* metric, which measures the percentage of branches of a model that are activated by a given test sequence, and the *Path coverage* metric which measures the percentage of paths that are traversed by a given test sequence, where a path is a sequence of branches that is traversed when going from the start of the model description to its end.
- **Bit coverage:** the model is proposed in [8][10]. The authors assume that each bit in every variable, signal or port in the model can be stuck to zero or one. The bit coverage measures the percentage of stuck-at bits that are propagated to the model outputs by a given test sequence.
- **Condition coverage:** the model is proposed in [8] and is intended to represent faults located in the logic implementing the control unit of a complex system. The authors assume that each condition can be stuck-at true or stuck-at false. Then, the condition coverage is defined as the percentage of stuck-at conditions that are propagated to the model outputs by a given test sequence. This model is used in [8] together with bit coverage for estimating the testability of complex circuits.

When the quality of test sequences is considered, the common assumption is that, given two test sequences, S_1 and S_2, of the same length, the better sequence is the one that attains the higher gate-level stuck-at fault coverage. Let us assume that it is S_1. The stuck-at fault model is commonly adopted as the reference metric for evaluating the goodness of vectors at the gate level. When the analysis of the two sequences S_1 and S_2 is moved to the high level, the adopted fault model should provide the same result, and thus the high-level fault coverage figure of S_1 should be higher than that of S_2. If this condition is not satisfied, the adopted high-level fault model is not suitable for representing meaningful information about the testability properties of test sequences.

Inspired by the above observation, we performed an intensive analysis of the available high-level fault models to evaluate how they compare with respect to the gate-level stuck-at fault model. The basic idea of the analysis process we developed is to fault simulate the *same* input sequence with two *different* models of the *same* circuit, a high-level model and the corresponding gate-level one, and then to compare the attained gate-level and high-level fault coverage figures. The analysis has been performed with 5 circuits from the High Level Synthesis'91 benchmarks suite; the circuits have been selected to represent different types, i.e., data dominated and control dominated ones. The test sequences have been generated both randomly and through a gate-level automatic test pattern generator (Synopsys `testgen`). As a mean for measuring the relationship between gate- and high-level coverage figures, we adopted the following correlation function

$$\frac{Cov(FC_{HL}, FC_{GL})}{s_{HL} s_{GL}}$$

where *Cov()* denotes the covariance operator, while FC_{HL} (FC_{GL}) is the set of high-level (gate-level) coverage figures we recorded by varying the input sequence length. Finally, s_{GL} (s_{GL}) is the standard deviation of the set of high-level (gate-level) coverage figures.

Table 1. Fault models comparison

Circuit type	Statement coverage	Bit+condition coverage
Data dominated	0.67	0.97
Control dominated	0.83	0.80

Table 1 summarizes the results we gathered during the analysis of fault models, where bit+condition coverage refers to the fault model obtained by combining bit coverage and condition coverage as proposed in [8]. The results show that, when the data-dominated circuits are considered, the statement coverage is poorly correlated with the gate-level fault coverage. Conversely, the bit+condition coverage shows a higher correlation. As far as the control-dominated circuits are concerned, we observed a good correlation between both statement coverage and bit+condition coverage with the gate-level stuck-at one. These results indicate that bit+condition coverage could be fruitfully used for evaluating the goodness of a test set at the high level. Given two different test sets, bit+condition coverage is able to estimate which of the two sets could attain the higher gate-level stuck-at fault coverage, *before* any synthesis step is performed. Moreover, the prediction about the gate-level fault coverage can be obtained *independently* on the optimization parameters used for driving the synthesis step. Although these preliminary results are promising, they pinpoint a limitation of the available fault models as far as control dominated circuits are considered. In this case, the correlation between high-level and gate-level fault coverage figures is still limited and should be improved.

Test Generation

This section presents two test vector generation environments we developed to evaluate the quality of test sequence generation in the early stages of the design cycle. The first one attacks test generation from a purely behavioral point of view and neglects all the information that may be available about the final implementation. The second approach is based on a hierarchical algorithm that takes into account also information about the modules used to implement the tested behavior.

High-level test generation

Following the high-level fault model analysis, where the bit+condition coverage has been selected as reference high-level fault model, we first developed a simple automatic test vectors generation (ATPG) algorithm whose task is to compute, at the high level, a set of test vectors able to attain high gate-level stuck-at fault coverage. The ATPG program is based on a Random Mutation Hill Climber (RMHC) algorithm that, given an initial randomly-generated solution, evaluates neighbor solutions in a complete random order until an improvement is found. When an improvement is found, the process is iterated over the new solution. Usually, a RMHC algorithm stops after a given amount of iterations. RMHC usually selects non-worsening solutions; however, to avoid an endless wandering in *mesas* (flat regions of the solution space), the RMHC algorithm implemented here accepts a new solution only if it represents an improvement of the current one.

In our algorithm a solution is a sequence of test vectors; one test vector is applied to the circuit inputs per clock cycle. Starting from an initial solution S, a new solution S' is computed by apply a random mutation operator. This operator supports two types of mutations: it complements one randomly selected bit within a randomly selected vector of S, or it linearly increases the number of vectors in S. The initial number of vectors in S is set to a user-specified value. The number of iterations the RMHC algorithm performs is also a user-specified parameter. During each iteration, a fault simulator supporting bit+condition coverage is used to measure the goodness of solutions the RMHC computes.

For the purpose of our experiments, we adopted three benchmarks:

- **BIQUAD:** it is an implementation of a bi-quadratic filter, whose equation is the following:

$$y_k = y_{k-1} \cdot a_1 + y_{k-2} \cdot a_2 + x_k \cdot b_0 + x_{k-1} \cdot b_1 + x_{k-2} \cdot b_2$$

- **FIR:** it is an implementation of Finite Impulse Response filter, expressed by the following equation (N=16):

$$y_k = \sum_{i=0}^{N} a_i \cdot x_{k-i}$$

- **TLC:** the benchmark implements a simple traffic light controller.

The above two filters represent data-dominated applications, while TLC represents an example of control-dominated applications. The benchmarks are available as VHDL and SystemC models that are coded according to the design rule of the Synopsys Behavioral Compiler tool. We synthesized two versions of each filter, one optimized for speed (BIQUAD 1 and FIR 1 in Table 2) and the other optimized for area (BIQUAD 2 and FIR 2 in Table 2). Due to the benchmark nature, only one implementation for TLC was synthesized.

The experimental results are reported in Table 2. We used stuck-at fault model for measuring the gate-level fault coverage and compared our results with the gate-level ATPG tool `testgen`. The results show that by reasoning only the behavior of a circuit we can generate useful test sequences. The fault coverage obtained by the high-level ATPG is comparable with coverage figures obtained at the gate-level, while test generation time is reduced significantly.

Table 2. Results of the high-level test generation algorithm.

Design	**High-level ATPG**			testgen		
	FC [%]	**Len [#]**	**CPU [s]**	FC [%]	Len [#]	CPU [s]
BIQUAD 1	**68.27**	**287**	**2,139**	37.06	154	10,817
BIQUAD 2	**86.94**	**287**	**2,139**	70.75	245	11,060
FIR 1	**91.27**	**2,413**	**1,157**	94.71	8,742	12,211
FIR 2	**89.77**	**2,413**	**1,157**	91.38	4,621	25,038
TLC	**80.88**	**110**	**225**	84.09	500	1,579

The results also show that in some cases there exists a gap between the fault coverage figures attained by test sequences generated purely on a high-level and those by the gate-level ones. Therefore a possibility to improve the high-level ATPG by integrating structural information to the test generation process has been investigated and a novel hierarchical test generation (HTG) algorithm has been developed.

Hierarchical Test Generation

The main idea of a HTG technique [11] is to use information from different abstraction levels while generating tests. One of the main principles is to use a modular design style, which allows to divide a larger problem into several smaller subproblems and to solve them separately. This approach allows generating test vectors for the lower level modules based on different techniques suitable for the respective entities. HTG has been successfully used until now for hardware test generation at the gate, logical and register-transfer levels. Our HTG operates on the behavioral level and employs constraint logic programming techniques with a decision diagram (DD) based representation [12].

Figure 1 depicts an example of DD, describing the behavior of a simple function. In this example, variable *A* will be equal to *IN1+2*, if the system is in the state *q=2* (Figure 1c). If this state is to be activated, condition *IN1³0* should be true (Figure 1.b) and in our terminology this is a path activation constraint for activating a path to the specified state (*q=2*). The DDs, extracted from a specification, will be used as a computational model in our HTG algorithm for symbolic path activation.

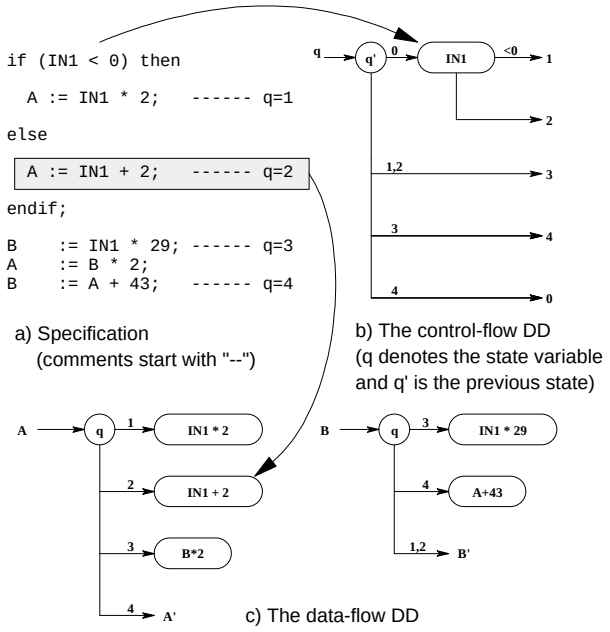

Figure 1. A decision diagram example

The HTG algorithm generates two types of tests, one for testing the behavior of the system and another for exploring information related to the final implementation of the system. The first set is generated from pure behavioral description based on certain code coverage metric [9], which has also been discussed earlier. This test set targets errors in branch selection (nonterminal nodes of the control-flow DD). During the second test generation phase the functional blocks (e.g., adders, multipliers and ALUs) composing the behavioral model are identified (terminal nodes of the data-flow DD), and suitable test vectors are generated for the individual blocks. During the block-level test generation phase each block is considered as an isolated and fully controllable and observable entity; and a gate-level test generation tool is used for this purpose. The test vectors generated for the basic blocks are then justified and their fault effects propagated in the behavioral model of the circuit under test. In this way we can incorporate accurate structural information into the high-level test pattern generation environment while keeping propagation and justification task still on a high abstraction level. In the following the test pattern generation algorithm is described in detail.

Conformity Test. For the nonterminal nodes of the control-flow DD, conformity tests will be applied. The conformity tests target errors in branch activation. For example, in order to test nonterminal node *IN1* (Figure 2), one of the output branches of this node should be activated. Activation of the output branch means activation of a certain set of program statements. In our example, activation of the branch *IN1<0* will activate the branches in the data-flow DD where *q=1 (A:=X)*. For observability the values of the variables calculated in all the other branches of *IN1* have to be distinguished from the value of the variables calculated by the activated branch. In our example, node *IN1* is tested, in the case of *IN1<0*, if *X¹Y*. The path from the root node of the control-flow DD to the node *IN1* has to be activated to ensure the execution of this particular specification segment and the conditions generated here should be justified to the primary inputs of the module. This process will be repeated for each output branch of the node. In the general case there will be *n(n-1)* tests, for every node, where *n* is the number of output branches.

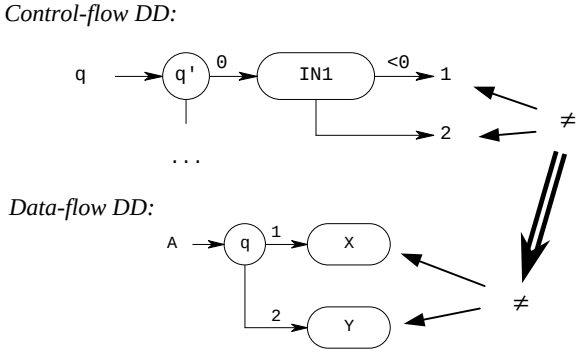

Figure 2. Conformity test

Testing Arithmetic Operators. One of the most important parameters guiding the synthesis process is the technology that will be used in the final implementation. By defining the technology, we can have information about the implementation of functional units that will be used in the final design. Our hierarchical test generation algorithm employs this structural information for generating tests. Tests are generated by cooperation of high-level and low-level test pattern generators as depicted in Figure 3. It is performed one by one for every arithmetic operator given in the specification.

We start by choosing a not tested operator from the specification and employ a gate level ATPG to generate a test pattern targeting structural faults in the corresponding functional unit. In our approach a PODEM like ATPG is used but in general case any gate-level test pattern generation algorithm can be applied. If necessary, pseudorandom patterns can be used for this purpose as well. The test patterns, which are generated by our current approach, can have some undefined bits (don't cares). As justification and propagation are performed at the behavioral level by using symbolic methods these

undefined bits have to be set to a defined value. Selecting the exact values is an important procedure since not all possible values can be propagated through the environment and it can therefore lead to the degradation of fault coverage. A test vector that does not have any undefined bits is thereafter forwarded to the constraint solver, where together with the environmental constraints it forms a test case. Solving such a test case means that the generated low-level test vector can be justified till the primary inputs and the fault effect is observable at the primary outputs. If the constraint solver can not find an input combination that would satisfy the given constraints, another combination of values for the undefined bits has to be chosen and the constraint solver should be employed again. This process is continued until a solution is found or timeout occurs. If there is no input combination that satisfies the generated test case, the given low-level test pattern will be abandoned and the gate-level ATPG will be employed again to generate a new low-level test pattern. This task is continued until the low-level ATPG can not generate any more patterns.

The HTG environment is depicted in Figure 3. Our HTG environment accepts as input a behavioral VHDL specification. The VHDL code is translated into the DD model, which is used as a mathematical platform for test generation, and later into a Prolog model, which is used by the constraint solver. In our approach we use a commercial constraint solver SICStus [13]. The HTG algorithm generates test cases and forwards them in form of constraints to the constraint solver, which generates the final test vectors. Propagation and justification of the gate-level test patterns are performed by the constraint solver as well.

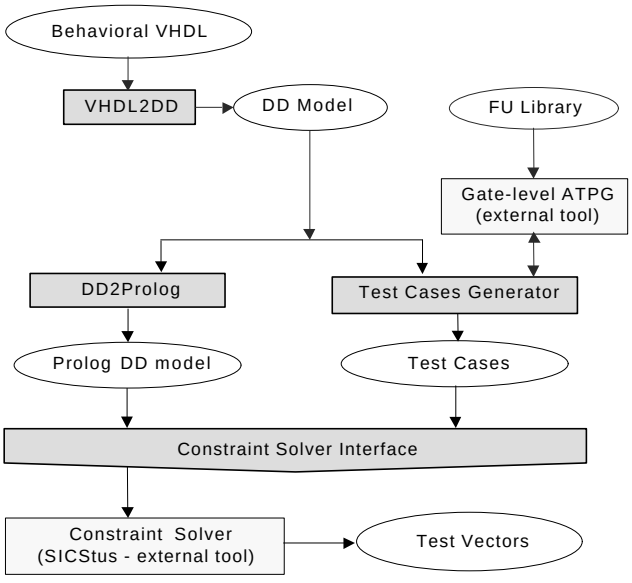

Figure 3. Our Hierarchical Test Generation Environment

We performed experiments on the DIFFEQ circuits taken from the High Level Synthesis'91 benchmark suite. The results are reported in Table 3, which shows that the test sequences the hierarchical test vector generator provides can be fruitfully used for testing stuck-at faults. These results show that when moving test vector generation toward lower levels of abstractions, where more detailed information about the tested circuits are available, the attained results in terms of fault coverage figures are improved. The fault coverage attained by the hierarchical ATPG is higher than that of the pure high-level ATPG, while the fault coverage working at the gate level is the highest. On the other hand, moving test generation towards the higher levels of abstraction has positive effects on the test generation time and on the test length that are both significantly reduced.

Conclusion

We have analyzed some high-level fault models in terms of the correlation they provide between high-level fault coverage and gate-level stuck-at fault coverage. In general, these fault models are not able to precisely foresee the gate-level fault coverage, but can be fruitfully exploited to rank test sequences according to their testability value to guide the generation of efficient test sequences. Thanks to this property, these fault models are suitable to be used within automatic test generation algorithms. Based on the adopted fault model we have demonstrated that test sequences generated from high-level descriptions provide good results when compared with gate-level ATPG in terms of required CPU time, attained fault coverage and obtained test length.

When moving from higher to lower levels of abstraction information will become available to the ATPG tool. A hierarchical test generation approach that takes into account information from several abstraction levels will therefore be able to generate test sequences with higher fault coverage than those of a pure behavioral test generator. Improvements in fault coverage can be obtained by integrating structural information coming from lower levels of abstractions, while still mainly working at the behavioral level for test vector justification and propagation. We have presented and demonstrated the efficiency of such a hierarchical test generation technique, which makes use of a constraint solving algorithm.

References

1 Moundanos, D., Abraham, J. A., Hoskote, Y. V., "A Unified Framework for Design Validation and Manifacturing Test", Proc. IEEE International Test Conference, 1996, pp. 875-884.

2. Ferrandi, F., Fummi, F., Sciuto, D., "Implicit Test Generation for Behavioral VHDL Models", Proc. IEEE International Test Conference, 1998, pp. 587-596.
3. Fallah, F., Ashar, P., Devadas, S., "Simulation Vector Generation from HDL Descriptions for Observability-Enhanced Statement Coverage", Proc. Design Automation Conference, 1999, pp. 666-671.
4. Chiusano, S., Corno, F., Prinetto, P., "Exploiting Behavioral Information in Gate level ATPG", JETTA:

The Journal of Electronic Testing: Theory and Applications, Kluwer Academic Publishers, No. 14, 1999, pp. 141-148.
5. Rudnick, E.M., Vietti, R., Ellis, A., Corno, F., Prinetto, P., Sonza Reorda, M., “Fast Sequential Circuit Test Generation Using High level and Gate level Techniques”, Proceedings IEEE European Design Automation and Test Conference, 1998, pp. 570-576.
6. Corno, F., Sonza Reorda, M., Squillero, G., “High level Observability for Effective High level ATPG”, Proc. 18th IEEE VLSI Test Symposium, 2000, pp. 411-416.
7. Beizer B., “Software Testing Techniques”, (2nd ed.). Van Nostrand Rheinold, New York, 1990.
8. Ferrandi, F., Ferrara, G., Scuito, D., Fin, A., Fummi, F., “Functional Test Generation for Behaviorally Sequential Models”, Proc. Design, Automation and Test in Europe, 2001, pp. 403-410.
9. Jervan, G., Eles, P., Peng, Z., “A Hierarchical Test Generation Technique for Embedded Systems”, Proc. Electronic Circuits and Systems Conference, 1999, pages 21-24.
10. Lajolo, M., Lavagno, L., Rebaudengo, M., Sonza Reorda, M., Violante, M., “Behavioral level Test Vector Generation for System-on-Chip Designs”, Proc. High level Design Validation and Test Workshop, 2000, pp. 21-26.
11. Murray, B. T., Hayes J. P., “Hierarchical Test Generation Using Precomputed Tests for Modules”, Proc. International Test Conference, 1988, pp. 221-229.
12. Ubar R., ”Test Synthesis with Alternative Graphs,” IEEE Design and Test of Computers, Vol. 13, No. 1, 1996, pp. 48-57.
13. SICStus Prolog User’s Manual, Swedish Institute of Computer Science, 2001.
14. Santos, M.B.; Goncalves, F.M.; Teixeira, I.C.; Teixeira, J.P., "RTL-based functional test generation for high defects coverage in digital SOCs", Proc. IEEE European Test Workshop, 2000, pp. 99-104.

Table 3. Comparing high-level, hierarchical and gate-level ATPGs

	High-level ATPG			**High-level Hierarchical ATPG**			**Gate-level ATPG `testgen`**		
	Gate-level FC [%]	Test Len [#]	CPU [s]	Gate-level FC [%]	Test Len [#]	CPU [s]	Gate-level FC [%]	Test Len [#]	CPU [s]
DIFFEQ 1	97.25	553	954	98.05	199	468	99.62	1,177	4,792
DIFFEQ 2	94.57	553	954	96.46	199	468	96.75	923	4,475