COTEST/D2

Report on automatic generation of test benches from system-level descriptions

Olga GOLOUBEVA, Matteo SONZA REORDA,
Massimo VIOLANTE
Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy

Contact person:

Matteo Sonza Reorda
Dipartimento di Automatica e Informatica
Politecnico di Torino
C.so Duca degli Abruzzi, 24
I-10129 Torino TO
Italy

Tel. +39 011 564 7055
Fax. +39 011 564 7099
E-mail: sonza@polito.it

Abstract

This document describes the process we followed for assessing the feasibility and effectiveness of high-level test vector generation. An experimental analysis of the available high-level fault models is first reported, whose purpose is to identify a reference fault model that could be fruitfully used for evaluating testability of circuits by reasoning on their behavior, only. A prototypical high-level fault simulation tool is also described, whose purpose is to support the fault models analysis. Finally, a test generation algorithm is presented that generates high quality test vectors by exploiting the selected fault model and the described high-level fault simulator.

Related documents

- COTEST Technical Annex
- COTEST Report D1: “Report on benchmark identification and planning of experiments to be performed”
- COTEST Report D3: “Report on Early DfT Support”

Table of Contents

1. Introduction 1
2. Fault model definition 1
    2.1. High-level fault models 1
    2.2. Fault models experimental analysis 2
        2.2.1. Behavioral synthesis 3
        2.2.2. High-level fault simulation 4
        2.2.3. Gate-level fault simulation 7
        2.2.4. Input selection process 8
    2.3. The considered benchmarks 8
    2.4. Fault model comparison 9
        2.4.1. Results for existing fault models 9
        2.4.2. A new fault model 10
3. High-level vector generation 10
    3.1. High-level ATPG 11
        3.1.1. The Simulated Annealing algorithm 12
    3.2. Gate-level ATPG 13
    3.3. Experimental results 13
4. Hierarchical vector generation 14
5. Conclusions 15
6. References 16

1. Introduction

This report describes the process we followed during the assessment of the feasibility and experimental evaluation of a high-level test sequence generation environment. The process is composed of two steps:

- *Fault model identification*: the purpose of this step is to select a suitable high-level fault model for estimating the testability of circuits by reasoning on their behavioral description, only. The process, that consists of an analysis phase, where the available fault models are analyzed, and a synthesis phase, where a fault model is selected, is described in section 2.
- *High-level test vectors generation evaluation*: the purpose of this step is to assess the effectiveness of a high-level test generation process. For this purpose we developed a very simple test generation tool based on the previously identified high-level fault model. This step is described in section 3.

Instrumental to the aforementioned steps is the definition of a suitable *Fault simulation tool* able to support us in the analysis of the fault models and during the high-level test generation process. The tool we prototypically developed is described in section 2.2.2.

2. Fault model definition

This section initially provides the reader with some background information about the most commonly adopted high-level fault models that can be found in literature. Then, the fault models analysis process is described, and finally the results we gathered are reported and commented.

2.1. High-level fault models

The most common high-level fault models proposed in literature as metrics to evaluate the value of test vectors resorting to behavioral level descriptions are the following:

- *Statement coverage*: it is a well-known metric in the software testing field [1] indented to measure the percentage of statements composing a model that are executed by a given test patterns. Further improvements of this metric are the *Branch coverage* metric, which measures the percentage of branches of a model that are executed by a given test patterns, and the *Path coverage* metric which measures the percentage of paths that are traversed by a given test patterns, where path is a sequence of branches that should be traversed for going from the start of the model description to its end.
- *Bit coverage*: the model proposed in [2][3]. 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 bit stuck-at that are propagated on the model outputs by a given test sequence.

- *Condition coverage*: the model is proposed in **[**2**]** and it is indented 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 condition stuck-at that is propagated on the model outputs by a given test sequence. This model is used in **[**2**]** together with bit coverage for estimating the testability of complex circuits.

For the analysis described in this report, we used the fault models summarized in the table 1.

Fault model name	**Description**
Statement coverage	It is the model proposed in [1]
Bit coverage	It is the model proposed in [2][3]
Condition coverage	It is the model proposed in [2][3]
Bit+condition coverage	This fault model combines the bit coverage with the condition coverage

Table 1: Adopted fault models

2.2. Fault models experimental analysis

As far as static test is addressed, the common assumption is that, given two test sequences S_1 and S_2 of the same length, the best sequence is the one that attains the highest gate-level stuck-at fault coverage. Let us assume that the best sequence is S_1. Stuck-at fault model is commonly adopted as the reference metric for evaluating the goodness of vectors at the gate level since it is able to model many types of physical defects and, at the same time, it is quite simple to deal with. Recent works, for example [4], show that in order to consider a larger set of defects, additional fault models should be considered. For example, defects leading to performance degradation mandates the execution of dynamic (possibly at-speed) testing. Furthermore, an accurate understanding of the relation between faults and defects can be obtained by computing the conditions under which defects are activated [5]. For the purpose of this project stuck-at faults are considered, only.

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 obtaining meaningful information about the testability properties of test sequences.

Given a correlation function that measures how good the match between gate-level and high-level fault coverage figures is, the ideal high-level fault model is the one that corresponds to the absolute maximum of the correlation function. In our work, we adopted the following correlation function, which measures the covariance of two data sets divided by the product of their standard deviations:

$$\rho_{X,Y} = \frac{Cov(X,Y)}{\sigma_X \cdot \sigma_Y}$$

The analysis process we adopted in this work is intended for measuring the fault coverage figures attained at the gate-level and the high-level one when the same test sequence is considered. These two sets are then correlated resorting to the adopted correlation function.

In order to measure the required coverage figures, we devised the experimental design flow reported in figure 1.

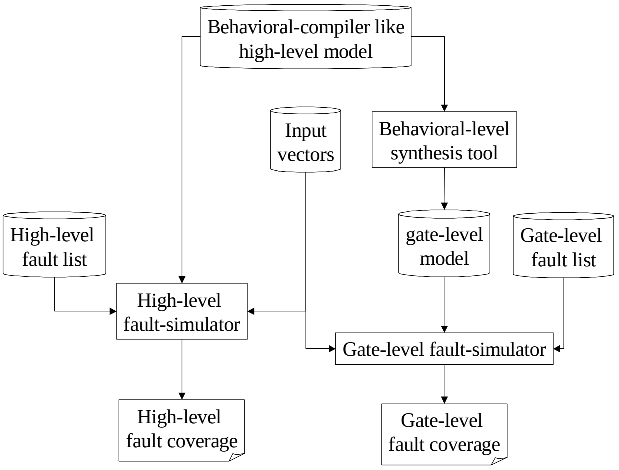

Figure 1: Experimental flow

Starting from a behavioral model of a circuit, two design flows are adopted. On the left-hand of figure 1 the flow for high-level fault coverage estimation is presented, where an ad-hoc developed high-level fault simulator is used to measure the fault coverage of the given input vectors set according to high-level fault models. The current implementation of the fault simulator, which is detailed in section 2.2.2, supports statement coverage, bit coverage and condition coverage, and it is able to address the simulation of either VHDL models or SystemC ones. On the right-hand of figure 1, the flow for evaluating the gate-level fault coverage of a given input set is presented.

The following sub-sections give details about the behavioral synthesis flow, the high-level and gate-level fault simulation procedures, and the input selection process we adopted.

2.2.1. Behavioral synthesis

We adopted a synthesis flow based on the Synopsys Behavioral Compiler tool [7] version 2001.08. The tool supports the synthesis of models coded according to the following assumptions:

- The model is composed of one entity.
- The entity is composed of one process.
- The process has an explicit clocking strategy, which is defined by using in the process the `wait` statement. Synthesis of processes embedding multiple `wait` statements is also supported.
- The process has a global synchronous reset signal.

The synthesis script we adopted is reported in figure 2.

```
analyze -s -f vhdl "./entity.vhd"

elaborate -s entity_name

current_design {"entity_name"}

create_clock -name "clk_name" -period CLOCK_PERIOD
-waveform { "0" CLOCK_PERIOD/2 } { "clk_name" }

current_design {"entity_name"}

bc_check_design

set_max_area 0

schedule -io_mode cycle_fixed /* CYCLE FIXED SCHEDULING */
schedule –io_mode superstate  /* SUPERSTATE SCHEDULING */
schedule –extend_latency /* EXPAND LATENCY FOR REDUCING AREA */

uniquify

compile –map_effort medium

ungroup –all –flatten

compile –map_effort medium
```

Figure 2: Synthesis script

Thanks to the Behavioral Compiler we are able to obtain gate-level implementations starting from purely behavioral models: the data path and the appropriate control unit are automatically derived by the synthesis tool.

A simple gate-level library composed of basic logic gates (AND, OR, etc.) and flip-flops is used for circuit representation.

2.2.2. High-level fault simulation

Fault model analysis mandates the availability of tools for estimating the fault coverage a given set of input vectors attains with respect to a reference high-level fault model. We developed a fault simulation approach that is independent from the adopted simulation environment and the language used from describing the simulated models. The approach consists of two phases:

1. The behavioral model of the circuit under analysis is first instrumented by adding suitable statements that fulfill two purposes:
   a. They alter the behavior of the model according to the supported fault models; namely bit and condition stuck-at fault models.
   b. They allow observing the behavior of the model to gather meaningful statistics; in particular, the number of executed statements and the number of reached `wait` statements are recorded.

   During this phase, the list of faults to be considered during fault simulation is computed and stored.
2. A given set of input vectors is applied to the inputs of the model resorting to the adopted simulation environment. During the execution of the model, first a simulation is performed without injecting faults, and the output trace of the circuit is recorded. Then, each fault in the previously computed fault list is injected and the model faulty behavior is observed. By comparing the model faulty trace with the fault-free one, we then compute the high-level coverage figure the test vectors set attains.

For the purpose of this project, we do not have developed any automatic model instrumentation tool. In the experiments we performed, instrumentation structures are manually inserted in the models before starting the simulation phase.

The adopted instrumentation structures are described in the following. Let us consider the VHDL code fragment reported in figure 3 as an example.

```
if A = ‘1’ then     -- Statement S1
  B <= not C;       -- Statement S2
end if;
```

Figure 3: Original code

The corresponding instrumented code is reported in figure 4.

```
if fault = ConditionStuckAtTrue then
  B <= not C;
else
  if fault = ConditionStuckAtFalse then
    ;
  else
    Monitor( “Executed Statement S1” );
    if A = ‘1’ then
      if fault = none
        Monitor( “Executed Statement S2” );
        B <= C;
      else
        if fault = BitStuckAt1
          B <= C or B_MASK;
        else
          B <= C and not B_MASK;
        end if;
      end if;
    end if;
  end if;
end if;
```

Figure 4: Supporting bit and condition stuck-at fault models

We add the signal `fault` to the port of the analyzed entity; the type of the added signal is an enumerative set whose possible values are: `ConditionStuckAtTrue`, `ConditionStuckAtFalse`, `BitStuckAt1`, `BitStuckAt0` and `none`. The first two values are used to support condition stuck-at faults, while the following two support bit stuck-at. The latter value is finally used for indicating that no faults are injected. A new port is then added for each of the model signals or variables where faults have to be injected (signal `B_MASK` in the example of figure 4). The port value is used to indicate which of the signal/variable bits should be set to 1 or 0.

Observation of the model is performed by inserting the call to the `Monitor()` function that writes on a trace file the current simulation time and the argument passed to the function. By parsing the trace file, one can count how many times a certain statement is executed and check whether statements do exist that are never executed.

Please note that, even if the instrumented model is far more complex than the original one (thus resulting in higher simulation times) the proposed approach can be adopted for model coded in any hardware description language. For the purpose of this project, we adopted this approach for performing fault simulation of model coded in VHDL and SystemC.

The architecture of the resulting high-level fault simulation tool is depicted in figure 5, where the following modules can be identified:

- The instrumented model which has been previously manually modified according to the described techniques.
- The simulation tool, which is either a VHDL simulator or a SystemC simulator.
- The experiment manger, which is in charge of:

a. Executing the simulator
b. Parsing the trace file for computing the statement coverage and the number or reached `wait` statements
c. Comparing the output trace produced by the faulty-free circuit with that of the faulty one.

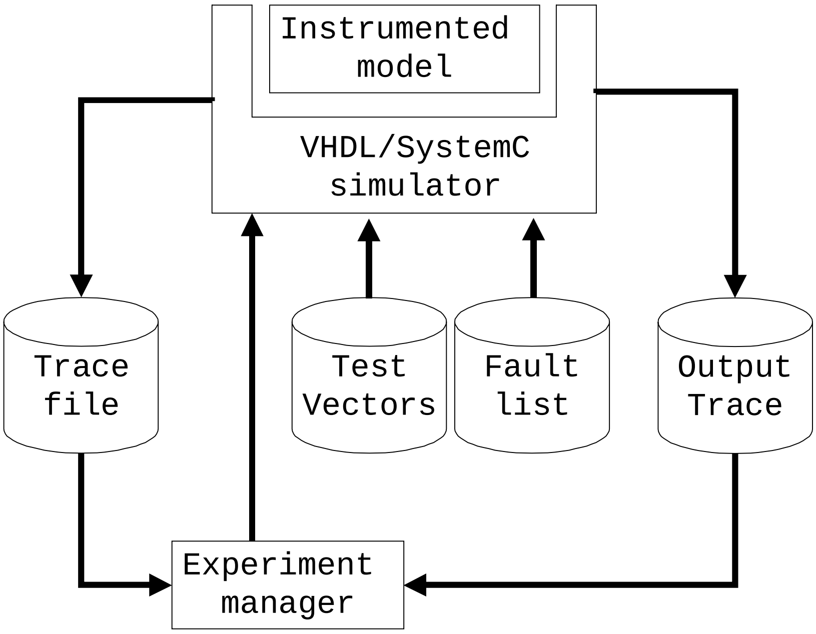

Figure 5: High-level fault simulation environment

The VHDL-based experiment manager amounts to 520 lines of C code, while the SystemC one amount to about 250 lines of C code.

2.2.3. Gate-level fault simulation

Gate-level stuck-at fault coverage is measured resorting to the Synopsys `faultsim` zero-delay fault simulator. Test vectors are formatted according to the configuration script reported in figure 6.

```
SEQUENCER
{
                SEQUENCE_LENGTH = 3;
                CLK_SIGNAL = "001";
                $STROBE_OPERATION = "010";
                RESET_SIGNAL = "111";
}
```

Figure 6: Test vectors formatting script

2.2.4. Input selection process

For the purpose of evaluating different high-level fault models, we adopted two types of vectors. Test vectors provided by a gate-level automatic test pattern generation program are used.

Moreover, for each of the analyzed circuits, we generated random vectors by adopting the following procedure:

1. Set the sequence length L to 10;
2. Generate 10 sequences of length L by using 10 different seeds;
3. Increase L by 10%
4. Repeat step 2 and 3 while L is less than 200.

2.3. The considered benchmarks

The benchmarks we adopted, that are detailed in [6], are shortly recalled here:

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

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

- *DIFFEQ*: it is the Differential Equation Solver model belonging to the High Level Synthesis’92 benchmarks suite. The model has been adapted to match the requirements of the Behavioral Compiler tool, while the data-path size has been set to 16 bits.
- *TLC*: the benchmark implements a simple traffic light controller.

The first two benchmarks represent examples of 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 [7]. We synthesized two versions of BIQUAD and DIFFEQ, one optimized for speed (BIQUAD 1/DIFFEQ 1 in table 2) and another optimized for area (BIQUAD 2/DIFFEQ 2 in table 2). Due to the benchmark nature, one implementation for TLC was synthesized, only.

The characteristics of the obtained circuits are reported in table 2, where the number of VHDL lines, the number of circuit input/output signals, the number of flip-flops and the number of gates are shown.

	VHDL lines	Inputs	Outputs	Flip-Flop	Gates
BIQUAD 1	93	9	17	125	6,043
BIQUAD 2	93	9	17	257	3,252
DIFFEQ 1	63	82	48	133	7,868

DIFFEQ 1	63	82	48	186	2,795
TLC	168	2	2	23	241

Table 2: Benchmark characteristics

2.4. Fault model comparison

For the adopted benchmarks, we compared gate-level suck-at fault coverage with statement coverage and the combination of bit coverage, condition coverage and bit+condition coverage.

2.4.1. Results for existing fault models

Table 3 reports the results we obtained when correlating the gate-level stuck-at with statement coverage, bit coverage, condition coverage and bit+condition coverage.

	Statement coverage	**Bit coverage**	**Condition coverage**	**Bit+condition coverage**
BIQUAD 1	0.63	0.97	0.60	0.97
BIQUAD 2	0.64	0.97	0.61	0.98
DIFFEQ 1	0.62	0.97	0.65	0.98
DIFFEQ 1	0.63	0.97	0.64	0.98
TLC	0.83	0.45	0.72	0.80

Table 3: Result for existing fault models: correlation between high-level fault coverage figures and gate-level ones

The results reported in table 3 show that, when data-dominated circuits are considered, the statement coverage and the condition coverage are poorly correlated with the gate-level fault coverage. Conversely, the bit coverage shows a higher correlation with the gate-level fault coverage for both the circuits implementations. Moreover, the bit+condition coverage shows the highest correlation figures for both the filter implementations.

As far as the TLC benchmark is considered, we observed a good correlation between both statement coverage with the gate-level stuck-at one, while the remaining fault models show lower correlation figures. Although for this benchmark the correlation coefficient is about 0.83, it is still far from the results we obtained for BIQUAD, thus suggesting that the available fault models are not sufficient for accurately estimate fault coverage while at the high level.

These results indicate that bit+condition coverage could be fruitfully used for evaluating the goodness of a test set while at the high level for data-dominated applications. 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* from the optimization parameters used for driving the synthesis step.

According to experiments for control-dominated applications, the available metrics seem insufficient and need further improvement. In particular, the gate-level model of a control-dominated circuit embeds several gates that do not correspond to any signal/variable or conditional statement of the behavioral model. These gates are produced by the synthesis tool for implementing the circuit control unit, and they are automatically derived after an analysis of the model clocking strategy. As a result, faults located in these gates can not be easily modeled resorting to the statements of the high-level model.

2.4.2. A new fault model

To cope with the control part of synthesized circuits we propose to combine the bit+condition coverage metric with a *state coverage* metric. The state coverage measures the percentage of the states of a model that are traversed during the simulation of a test set. This information can be easily extracted by the trace file by counting the number of different `wait` statements that the given test pattern allows to reach.

In our experimental analysis we observed that by considering the extended fault model, which we refer to as bit+condition+state coverage, the correlation between high-level fault model and the gate-level one is increased. In particular, the benefits stemming from the new fault model have been observed for the control-dominated benchmark.

	Bit+condition coverage	**Bit+condition+state coverage**
BIQUAD 1	0.97	0.97
BIQUAD 2	0.98	0.98
DIFFEQ 1	0.98	0.98
DIFFEQ 1	0.98	0.98
TLC	0.80	0.86

Table 4: Results for the new fault model

3. High-level vector generation

The experimental analysis described in the previous section allowed us to identify a high-level fault model that can be used to estimate the testability of a circuit without knowing its actual gate-level implementation. The fault model is indeed based on measurements that can be performed on a purely behavioral model of the analyzed circuit.

To assess the effectiveness of the proposed high-level fault model in driving a high-level test generation procedure, we developed a prototypical Automatic Test Pattern Generation (ATPG) tool. Sub-section 3.1 describes the tool that adopts bit+condition+state coverage as metrics of the goodness of test sequences and exploits the high-level fault simulator described in sub-section 2.2.2. It is worthwhile to remark that the purpose of the developed ATPG is to assess the feasibility of a high-level test generation process. The developed tool is therefore not optimized and kept as simple as possible.

To validate the effectiveness of the developed high-level ATPG, we compared the results it provides with those provided by the Synopsys `testgen` tool working on the corresponfing gate-level description. Section 3.2 reports details on the adopted gate-level test generation procedure, while sub-section 3.3 reports the figures we gathered during our experiments.

3.1. High-level ATPG

The test generation algorithm we developed is based on a Simulated Annealing algorithm.

The architecture of the ATPG we developed is described in figure 7.

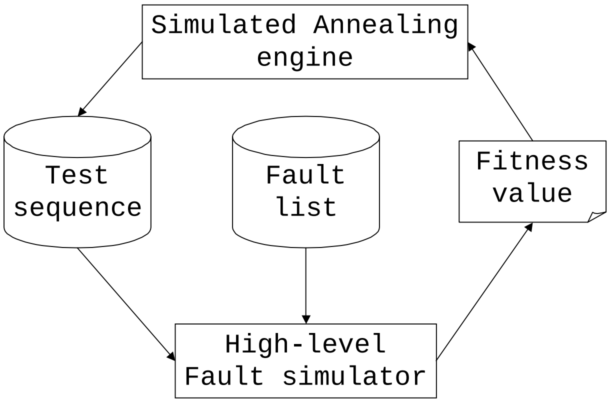

Figure 7: Architecture of the developed ATPG

Two main modules compose the developed ATPG:

- *Simulated Annealing engine*: this module takes care of generating the initial solution and controlling its refinement. Details on this module are reported in sub-section 3.1.1.
- *High-level fault simulator*: it is the tool described in sub-section 2.2.2.

The communication between the two modules takes place in the following way:

- The Simulating Annealing engine sends the solution to the simulation tool to evaluate its goodness according to the selected fitness function. For the purpose of the current implementation of the ATPG algorithm, the fitness function is the bit+condition+state coverage.
- The fault simulation tool applies the input sequence to the model inputs and measures the fault coverage figure it attains, and then commits it to the Simulated Annealing engine.

3.1.1. The Simulated Annealing algorithm

The selection process is based on a Simulated Annealing algorithm. Simulated Annealing was introduced in [8] and is used to give approximate solutions to very large combinatorial problems. The technique originates from the theory of statistical mechanics and is based upon the analogy between the annealing process of solids and the solving procedure for large combinatorial optimization problems.

A *configuration* C_i is defined as a solution for a given combinatorial problem. Each configuration C_i has an associated *cost* E_i given by a *cost function* $E_i=E(C_i)$. The optimization goal is to find the *configuration* that minimizes the given *cost function* E.

The *neighborhood* of a configuration C_i is defined as the set of configurations that can be obtained from C_i by applying small *perturbations*.

The basic version of a Simulated Annealing algorithm [9] can be formulated as follows:

- Starting from an initial configuration C_0, a sequence of iterations is performed;
- In each iteration a new configuration C_{i+1} is selected in the neighborhood of C_i and the variation in the cost function

  $$\Delta E_i = E(C_{i+1}) - E(C_i)$$

  is computed;
- If ΔE_i is negative, the transition from C_i to C_{i+1} is unconditionally accepted; if the cost function increases, the transition is accepted with a probability based upon the Boltzmann distribution

  $$P_{acc}(\Delta E_i) = \exp(\, -\Delta E_i / kT \,)$$

  where k is a constant and the temperature T acts as a control parameter;
- The temperature is gradually decreased throughout the algorithm from a sufficiently high starting value (i.e., a temperature where almost every proposed transition, positive or negative, is accepted) to a *freezing* temperature, where no further changes occur.

The set of parameters controlling the temperature decrement (initial temperature, stop criterion, temperature decrement between successive stages, number of iterations for each temperature value) is called the *cooling schedule*.

Consequently, the key elements for the implementation of Simulated Annealing are:

- *The definition of the initial configuration*. For the sake of our ATPG is a randomly generated sequence of vectors. The number of vectors in the sequence is a user specified parameter.
- *A generation mechanism*, i.e., the definition of a neighborhood on the configuration space and a perturbation operator exploring it. A neighbor solution is computed by resorting to two mutation operators. The first one complements one randomly selected

bit in the test sequence, the second one adds one vector to the test sequence. The first operator has an activation probability of 90% while that of the second one is 10%.

- *The choice of a cost-function*. Is it the bit+condition+state coverage measured by the high-level fault simulator.
- *A cooling schedule*. We adopted an exponential decrease in the value of the temperature *T* starting from the value of 250 Kelvin.

3.2. Gate-level ATPG

Gate-level test vectors generation is performed resorting to the Synopsys `testgen` tool. Please note that we are dealing with the generation of test vectors assuming that the circuit is not provided with any design-for-testability structure, such as full or partial scan: we used `testgen` as a purely sequential ATPG tool.

The constraint file reported in figure 8 is used for driving the gate-level test generation phase.

```
$constraints {
        $testgen {
                $pat CLK_SIGNAL = 001;
                $pat RESET_SIGNAL = CHH;
                $default = CHH;
                $pat $STROBE_OPERATION = 010 ;
        }
}
```

Figure 8: Gate-level ATPG configuration file

3.3. Experimental results

We compared the results the gate-level ATPG and the high-level one produced. All the experiments have been performed on a Sun Blade 1000 workstation running at 775 MHz and equipped with 1 GB of RAM.

Table 5 reports the attained results for the considered benchmarks. By observing the figures we gathered, the following considerations arise:

- As far as the data-dominated models are considered, the high-level ATPG we developed together with the underlying high-level fault model we proposed can be effectively used for generating test vectors while at the high level. In particular, by resorting to an abstract model of a circuit where its behavior is considered only, we were able to efficiently compute high quality test sequences with respect to a commercially available gate level tool: when simulated, high-level generated vectors show a high stuck-at fault coverage, and in the case of the BIQUAD benchmark the recorded figures are higher than those of the gate level ATPG by a factor of 31%

(BIQUAD 1) and 16% (BIQUAD 2). Moreover, by resorting to an abstract circuit model, we can drastically reduce the time for fault simulating test sequences. The CPU time our algorithm used for performing test generation is indeed about 5 times less than that the gate-level tool spent.

- As far as the control-dominated model is considered, we observed that the fault coverage figure attained by `testgen` sequences is higher than that of the high-level generated ones. This result is motivated by the low correlation still existing between the high-level fault model and the low-level one. On the other hand, the benefits of running test generation on high-level models becomes evident when comparing the length of the attained sequences and the test generation time. For both this parameters, the high-level test generation tool prevails over the gate level one: the test set it provided is indeed about 3 times shorter then the gate-level one and the CPU is about 10 times lower than that of `testgen`.

	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
DIFFEQ 1	97.25	553	954	99.62	1,177	4,792
DIFFEQ 2	94.57	553	954	96,75	923	4,475
TLC	74.48	23	10	80.67	77	98

Table 5: ATPG results

4. Hierarchical vector generation

The experimental results reported in section 3.3 show that by reasoning only on the behavior of a circuit we can generate useful test sequences. The results also show that in some cases a gap between the fault coverage figures attained by high-level generated sequences and the gate-level ones still exists. In this section, a technique developed at the LiU for further improving the test generation capabilities by mixing high- and low-level information is shortly described. The approach is based on taking into account structural information during the test generation process. The technique has been developed as part of a testability analyzer tool intended for driving a high-level testability structure insertion process and it is detailed in the COTEST deliverable D3 [11].

The hierarchical test vector generation approach is based on the following steps:

- The functional blocks (e.g., adders, multipliers, …) composing the behavioral model are first identified, and then suitable test vectors are generated for these blocks. During the test generation phase each block is considered as an isolated and fully controllable and observable entity; a gate-level test generation tool is used for this purpose.

- The previously identified set of vectors is justified and fault effects are propagated to the behavioral model of the circuit under test.

The input stimuli the hierarchical test vectors generator tool provides can be fruitfully used for testing stuck-at faults, as the results reported in the following confirm. Table 6 shows the fault coverage figures the hierarchical ATPG attains when generating vectors for the two implementations of the DIFFEQ benchmark. For the sake of comparison, we also report the results attained by the high-level ATPG tool described in section 3.1 as well as the `testgen` ones.

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. Indeed, as table 6 shows, the fault coverage attained by the hierarchical ATPG is higher than that of the high-level ATPG, while the fault coverage `testgen` attains (working at the lowest level of abstraction) is the highest. Conversely, moving test generation at the higher levels of abstraction has positive effects on the test generation time and on the test length that are both significantly reduced.

	High level ATPG			**Hierarchical ATPG**			`testgen`		
	FC [%]	Len [#]	CPU [s]	FC [%]	Len [#]	CPU [s]	FC [%]	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

Table 6: Comparing ATPG results at different levels of abstraction

5. Conclusions

Given the assessment criteria described in the project technical annex [10], the results we gathered show that high-level test generation is effective and viable for data-dominated applications: in our experiments working on high-level descriptions allowed to reduce the test generation time by a factor of 5 and, in some cases, to increase the attained fault coverage by more than 5%.

As far as control-dominated applications are considered, the results we gathered show that the available high-level fault models do not allow high-level test generation to clearly prevail over gate-level one: while required CPU time is strongly reduced, and the test length is also lower, the attained fault coverage is lower when working on high-level descriptions. At the same time, the results are encouraging, since the sequences generated at the high-level are very compact and require a much lower CPU time than those generated from gate-level descriptions.

Further improvements in the attained fault coverage figures can be obtained by integrating structural information, coming from lower level of abstractions, while still mainly working at the behavioral level for vector justification and propagation.

6. References

[1] B. Beizer, “Software Testing Techniques”, (2nd ed.), Van Nostrand Rheinold, New York, 1990

[2] F. Ferrandi, G. Ferrara, D. Scuito, A. Fin, F. Fummi, “Functional Test Generation for Behaviorally Sequential Models”, Proc. Design, Automation and Test in Europe, 2001, pp. 403-410

[3] M. Lajolo, L. Lavagno, M. Rebaudengo, M. Sonza Reorda, M. Violante, “Behavioral level Test Vector Generation for System-on-Chip Designs”, Proc. High Level Design Validation and Test Workshop, 2000, pp. 21-26

[4] M. B. Santos, F. M. Goncalves, I. C. Texeira, J. P. Texeira, “RTL-based functional test generation for high defects coverage in digital SOCs”, Proc. IEEE European Test Workshop. 2000, pp. 99-104

[5] W. Kuzmicz, W. Pleskacz, J. Raik, R. Ubar, “Defect-oriented fault simulation and test generation in digital circuits”, Proc. Int. Symposium on Quality Electronic Design, 2001, pp. 365-371

[6] M. Sonza Reorda, M. Violante, “Report on benchmark identification and planning of experiments to be performed”, COTEST Report D1

[7] Behavioral Compiler VHDL Modelling Guide, Ver. 2001.08, Synopsys

[8] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, “Optimization by simulated annealing,” Science, 220, pp. 671-680, 1983

[9] P. van Laarhoven, E. Aarts, “Simulated Annealing: Theory and Applications”, D. Reidel Publishing Co., 1987

[10] COTEST Technical Annex

[11] P. Eles, G. Jervan, Z. Peng, “Report on Early DfT Support”, COTEST Report D3