VHDL Testability Analysis based on Fault Clustering and Implicit Fault Injection

F.S.Bietti F.Ferrandi F.Fummi D.Sciuto
Dip. di Elettronica e Informazione
Politecnico di Milano, 20133 Milano, ITALY

Abstract

Testability analysis of VHDL sequential models is the main topic of this paper. We investigate the possibility to obtain information about the testability of a sequential VHDL description before its actual synthesis. The analysis is based on an implicit fault model that injects faults into a BDD based description extracted from the VHDL representation. Such an injection is related to the original VHDL representation thus allowing the identification of potential testability problems before RTL and logic synthesis. Fault injection is performed efficiently by exploiting the concept of fault clustering, that is, the possibility of grouping faults and analyzing them concurrently. The proposed methodology is applied to benchmarks for efficiency evaluation and to a real VHDL description.

1 Introduction

Algorithms concerning the testing field are based on models which abstract the behavior of physical defects. Such models include fault models, error models or failure models depending on the abstraction level to which they refer [1]. The use of models sensibly reduces the complexity of all testing algorithms since it decreases the number of different entities which must be manipulated. For instance, some defects can correspond to few faults which may or may not produce a single error. *Functional* fault models belong to this class and they have been used for two main purposes:

- The identification of *testability problems* in order to avoid the RTL and logic synthesis of specifications which show testability problems independently of their actual synthesis [16, 15, 11].
- The generation of *fault independent* test patterns which allow the detection of the majority of gate-level faults and which preserve this property even considering different implementations of the same circuit [4, 5, 12].

However, the complexity of testing problems remain high even restricting the attention to single fault or error, for instance, by adopting the well known single stuck-at fault model [1] or single transition fault model [5]. There is thus the need of using implicit techniques, based on binary decision diagrams (BDDs) [3], to analyze complex descriptions that cannot be explicitly managed by using state transition tables or graphs [4, 5, 12, 15]. Moreover, hardware description languages (e.g., VHDL or Verilog) are widely used for the specification and automatic synthesis of a device, thus a useful testability measure must be related to a HDL description of a device. We oriented this paper to the analysis of sequential VHDL descriptions.

This paper analyzes the problem of testability analysis by starting from the test pattern generation approach presented in [8]. Testability measure is based on the actual fault coverage obtained by generating test sequences based on a functional fault model that is related to gate-level (stuck-at) faults. The aim is the prediction of the actual stuck-at fault coverage before RTL and logic synthesis. For this purpose, fault clustering can be used to reduce the number of analyzed errors by discarding errors which have equivalent behaviors in order to identify a subset of the total number of modeled errors which ensures a prediction of the stuck-at fault coverage close to the estimation obtained by considering all errors. The proposed functional fault model improves the previous work [8] on the following aspects:

- Concurrent analysis of groups of faults (*clustering*) has been implemented to make the testability analysis more efficient.
- Some relations between *functional faults* and *VHDL faults* have been identified to improve the accuracy of the obtained testability measure.
- The set of analyzed *functional faults* has been oriented to the RTL and logic synthesis tools, since adopted synthesis algorithms sensibly impact on the testability of the generated circuits.

The rest of the paper is organized as follows. Section 2 presents the basic models adopted in this paper and in particular the translation algorithm which is able to convert a VHDL representation into the equivalent BDD based description. Section 3 is devoted to the description of the adopted functional fault model based on BDDs and of the identified relations between VHDL faults, functional faults and stuck-at faults. The last section presents some experimental results on the efficiency of faults clustering and some preliminary results on the effectiveness of the proposed functional fault model for the prediction of stuck-at fault coverage.

0-8186-8409-7/98 $10.00

2 Basic Models

This section introduces the basic concepts to describe the VHDL-level fault model used in this paper to implement the proposed testability estimator. The model is based on a BDD representation of the behavior of each device. Thus, we first introduce the BDD based description of a FSM, then the technique to extract the BDD representation from a VHDL description at the register-transfer (RT) level.

2.1 Basic Definitions

Let us restrict our attention to sequential circuits representing medium and large size controllers. Let a FSM M be the 5-tuple

$$M = (X, Z, S, S^0, R)$$

where X is the input alphabet, Z is the output alphabet, S is the finite set of states, S^0 is the reset state, and $R \subseteq S \times X \times S \times Z \rightarrow \{0,1\}$ is the *global relation*. We have that the characteristic function $R(x,z,s,t) = 1$ if and only if, under input $x \in X$, the FSM makes a transition from present state $s \in S$ to next state $t \in S$ outputting $z \in Z$. We represent this characteristic function by using a BDD based description. A FSM can be represented by a *state transition graph* (STG), whose vertices are elements of $s \in S$ and edges are labeled with pairs $(x,z) \in X \times Z$. Input symbols X are coded by I input variables $i_1, \cdots, i_I$ and output symbols Z are coded by O variables $o_1, \cdots, o_O$.

Let M_f be the FSM representing the behavior of M affected by fault f. The method for generating M_f depends on the abstraction level of fault f and it is described in the next section.

A *test sequence* for fault f is a sequence of input vectors such that, when applied to machines M and M_f (both started in their reset states), produces two different output vectors for M and M_f. By concurrently traversing both M and M_f starting from their reset states, it is possible to generate (if any) a test sequence.

2.2 VHDL to BDD Translation

We propose here to obtain the *global relation* of the analyzed circuit through direct translation of a VHDL source code. There are three main reasons advising the direct translation of VHDL into BDDs to perform testability analysis.

- Testability information becomes available before RTL synthesis. The designer can evaluate the testability problems, concerning the specification of VHDL entities, before their RTL synthesis. It is fundamental to modify for testability a design from the early stages of its specification.
- VHDL synthesis tools infer some memory elements during RTL synthesis that are not concerned with the actual behavior. Such memory elements increase the number of states of the implemented controller and also the complexity of the corresponding BDD representation. For instance, let us consider the simple VHDL code reported in Figure 1, describing the behavior of a synchronous controller when it is in state *p0*. The output signals `end_master`, `valid_pol` and `start_r` are assigned into a synchronous process, thus they must be connected to flip-flops. Moreover, `end_master` is not assigned in the `ELSE` branch of the condition, thus it requires a *default* value to be synthesized. Consequently, the synthesis tool inserts a *muxed-flip-flop* onto such an output port. It is evident that the implemented FSM has higher number of states with respect to the specification, and most of them are equivalent. Thus, the global relation extracted from the implementation would have an unnecessarily high number of state variables.
- Unspecified transitions are not included in the global relation. During the extraction of the global relation of a sequential circuit from its implementation, even transitions outgoing from unspecified states are included. In fact, all combinations of input and present state variables are implicitly taken into account to identify the FSM's transitions. On the contrary, the global relation of a controller includes only specified transitions if it is directly constructed from a VHDL specification. A lower number of transitions usually implies a smaller BDD.

```
.....
CASE state IS
   WHEN p0 =>
      IF start_master = '1' THEN
         state <= refresh;
         end_master <= '0';
         valid_pol <= '0';
         start_r <='1';
      ELSE
         state <= p0;
         valid_pol <= '1';
         start_r <= '1';
      END IF;
.....
```

Figure 1: VHDL Description of a Part of a Controller.

The technique we propose to translate a VHDL description specifying a controller into the corresponding relation represented as a BDD starts by analyzing the VHDL architecture associated with the entity of the FSM. Two types of analysis are performed. First, a state identification is carried out. In fact, considering the execution flow of the sequential process, a signal or variable belongs to the state of the FSM when it is read before any write or initialization instruction. When the VHDL description considers both the FSM and the associated data-path also the data-path registers are considered as state variable in the global relation. After state identification, the global relation is computed. The style of the finite state machine considered presents a process with a conditional structure. Details about the translations of VHDL into BDDs can be found in [2].

3 Fault Model

The adopted functional fault model is based on the modification of the *global relation* $R(x,z,s,t)$ representing the functional description of the circuit, directly extracted from its VHDL description.

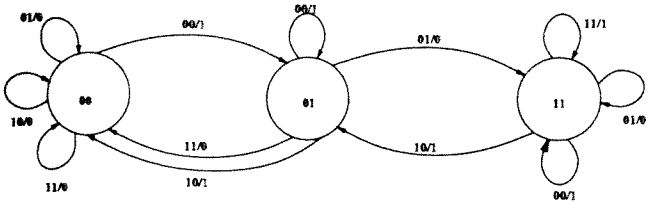

Figure 2: Simple FSM.

Let us consider, for instance, the simple FSM reported in Figure 2. Its global relation is explicitly represented in Table 1.

x_1x_2	z_1	s_1s_2	t_1t_2	x_1x_2	z_1	s_1s_2	t_1t_2
00	1	00	01	01	0	00	00
10	1	00	00	11	0	00	00
00	1	01	01	01	0	01	11
10	1	01	00	11	0	01	00
00	1	11	11	01	0	11	11
10	1	11	01	11	1	11	11

Table 1: Global Relation of the simple FSM reported in Figure 1.

3.1 Functional Fault Model

The *global relation* $R(x, z, s, t)$ must be partitioned at first into N single-output n-input Boolean functions $R_i(x, s)$, where $N = |t| + |z|$ and $n = |x| + |s|$. A minimal two-level implementation of R_i is composed of *essential primes* and *primes* that together cover the *ON*-set of the function with minimum cost. Let p_e be the set of essential primes, $|p_e|$ its cardinality, and k_e the total number of its literals. Similarly, let p be one set of primes required to complete the minimum cost cover, $|p|$ its cardinality, and k the total number of its literals.

All the stuck-at faults of the implementation of function R_i are guaranteed to be equivalent to the faults belonging to the following classes [1].

1. SA-1 faults on all literals of each prime in $p_e \cup p$ (i.e., they are equivalent to SA-1 faults on the *and* and *or* gates).

2. SA-0 faults on *one* literal of each prime in $p_e \cup p$ (i.e., they are equivalent to SA-0 faults on the *and* and *or* gates).

3. SA-0 and SA-1 faults on all input variables (i.e., they represent stuck-at faults on the *inverter* gates by assuming that all input variables are in complemented form for at least one prime in $p_e \cup p$).

The upperbound U_g on the total number of faults is $U_g = k_e + k + |p_e| + |p| + 2n$.

Clearly, faults corresponding to essential primes are present in every two-level implementation of the Boolean function, independently of the other primes selected to complete the cover (i.e., the primes in p). A necessary condition to test all stuck-at faults of every two-level implementation of R_i is the test of faults corresponding to the essential primes.

To reduce the number of modeled faults, only faults on essential primes are considered. This simplification extremely reduces the computation time for the reasons described in the next section. Therefore, the number L_g of faults that will be considered is $L_g = k_e + p_e + 2n$.

The faulty global relation, R_F, is then determined based on the set of essential primes p_e, and on the previously described three classes of faults. The following three different strategies for fault insertion are applied in relation to the type of fault.

- SA-1 faults on all literals of each essential prime in p_e..
  Let $ip(x_1, \cdots, x_l, s_1, \cdots, s_m) \in p_e$ be an essential prime, and $b_i : b_i = x_i | b_i = s_i$.
  $\forall b_i : ip(x, s)_{b_i} \neq ip(x, s)_{\overline{b_i}}$ (i.e., $ip(x, s)$ depends on b_i) then
  the faulty prime $ip_F(x, s) = ip(x, s)_{b_i}$ and
  the faulty relation $R_{i_F}(x, s) = R_i(x, s) \cdot \overline{ip(x, s)} + ip_F(x, s)$
  (i.e., the fault-free essential prime is replaced by the corresponding faulty prime).

- SA-0 faults on *one* literal of each essential prime in p_e.
  Let $b_i : b_i = x_i | b_i = s_i$.
  $\exists b_i : ip(x, s)_{b_i} \neq ip(x, s)_{\overline{b_i}}$ then
  the faulty prime $ip_F(x, s) = ip(x, s)_{\overline{b_i}}$ and
  the faulty relation $R_{i_F}(x, s) = R_i(x, s) \cdot \overline{ip(x, s)} + ip_F(x, s)$.

- SA-0 and SA-1 faults on all input variables.
  Let $b_i : b_i = x_i | b_i = s_i$.
  $\forall b_i : R_i(x, s)_{b_i} \neq R_i(x, s)_{\overline{b_i}}$ then the faulty relation
  $R_{i_F}(x, s) = R_i(x, s)_{\overline{b_i}}$ for SA-0 fault and
  $R_{i_F}(x, s) = R_i(x, s)_{b_i}$ for SA-1 fault.

Finally, the global faulty relation, $R_F(x, z, s, t)$, is reconstructed by composing all computed faulty relation $R_{i_F}(x, s)$.

3.2 Functional Fault Clustering

Even if the proposed functional fault model deals with essential primes only, their explicit generation is a CPU intensive task for medium and large size controllers. Thus, a fast implicit method is necessary and we adopted the algorithm described in [7]. However, this approach becomes extremely long if the generation of all implicants would be necessary for the identification of essential primes. Fortunately, the use of *zero-suppressed BDD* (ZDD [10]) allows the implicit generation of essential primes without the enumeration of all implicants. Furthermore, the number of essential primes is usually small. Thus, after their implicit generation, essential primes can be explicitly enumerated by recursively visiting the ZDD graph. Moreover, to further speed up the process, a cluster of faulty primes can be extracted and inserted into the global relation in one step.

Fault clustering is possible by simply adding f variables to the faulty global relation ($R_F(x, z, s, t, f)$)

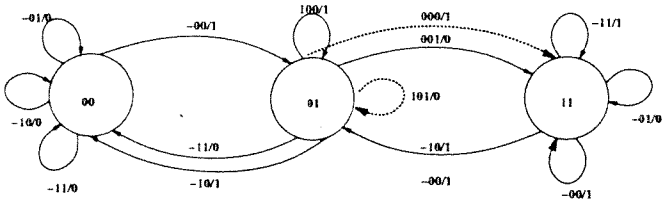

Figure 3: Faulty FSM with a cluster of two functional faults.

to discriminate the different faulty behavior of each fault in the cluster. For instance, consider the essential prime $\overline{x_1}x_2s_2$ of the simple FSM reported in Figure 2. The injection of the two functional faults SA-1 on x_2 and SA-0 on the entire prime produces the faulty global relation graphically shown in Figure 3. Dotted edges are faulty edges with respect to the fault-free FSM (see Figure 2). The faulty behavior referred to each fault in the cluster is discriminated by the first bit of the edge label that represents the added f variable (F added variables allow the clustering of 2^F faults).

3.3 Faults Manipulation

The fault model previously described can produces a functional fault coverage that can be not sufficiently related to the stuck-at fault coverage obtained after synthesis. Some manipulation criteria must be applied to the faults list in order to obtain an effective testability estimation. Such criteria can be summarized as follows.

3.3.1 Cyclic cores

Some Boolean functions have a cyclic core [9] that represents a relevant part of the ON-set. In this case the number of faults modeled by essential primes is a fraction of the total number of faults, thus the testability estimation can be affected by an unacceptable error. In this case, the cyclic core is removed by randomly extracting a prime (non essential) and new essential primes are identified. Faults are thus injected into essential primes and into so called *secondary* essential primes [9].

3.3.2 Multiple outputs

Essential primes are computed by independently analyzing each function $R_i(x, s)$. This operation is acceptable if faults are injected into functions corresponding to primary outputs, since even a single faulty output is sufficient to detect a fault and the observation of the fault effect on more outputs does not change the nature of the fault (testable/untestable). On the contrary, if function $R_i(x, s)$ represents a next state function, the modification of a single next-state bit, with respect to the modification of multiple next-state bits, can sensibly decrease the testability measure. In fact, faults affecting only one next-state bit are difficultly propagated to the primary outputs [6]. To take into account this observation a further analysis is implemented during the construction of the fault list. Primes related to next-state functions are injected into *all* next-state functions even if they are not *essential* for such functions. This operation takes into account the logic sharing of a typical multi-level implementation and produces a set of functional faults more related to stuck-at faults.

3.3.3 Don't care functionalities

A fault must be activated to be detected and the fault effect must be propagated to a primary output. A justification sequence must exist allowing to reach the activation state of the fault. Faults which cannot be activated are untestable and they are also called *sequentially-non-excitable* faults [6]. All faults which can be activated from unreachable states only belong to this group. The *don't care* functionality of a circuit represents the set of behaviors which cannot be activated since they require to start from unreachable states. The gate-level description of a device must *ideally* not include logic implementing *don't care* functionalities only, since it is a waste. However, commercial VHDL synthesizers (Mentor and Synopsys) are able to identify unreachable states only if they are *explicitly* enumerated (e.g., the states of a controller), but they are not able to identify unreachable states concerning *implicit states*.

```
...
CASE state IS
   ...
   WHEN iteration =>
       IF (counter < MAX) THEN
           counter := (counter + 1);
           state <= iteration;
       ELSE
           state <= operation;
       END IF;
...
```

Figure 4: VHDL Description with *implicit* and *explicit* states.

For instance, let us consider the small part of VHDL code reported in Figure 4. Variable `state` represents the actual state of the FSM and it is explicitly enumerated. It can assume 24 different values labeled by names (e.g., `iteration`). On the contrary, variable `counter` is defined as an integer subset from 0 to `MAX-1` and it represents an implicit counter, that counts from 0 to `MAX-1` and returns to 0 when the reset signal (not shown in Figure 4) is asserted. After states coding, there are 8 unused state codes ($32 - 24 = 8$) corresponding to the *don't care* part of the specification. Such states are recognized by the synthesizer and during RTL synthesis no logic is generated based on such states, that is, next-state and output functions are described at the RT level only for the explicitly enumerated states. For this reason, it is extremely likely that unused state codes are not necessary to detect faults and do not generate *sequentially-non-excitable* faults. On the contrary, the behavior of the circuit concerning implicit states (as the `counter` of this example) if specified for all state values even if some states are actually unreachable. For instance, the RTL synthesis instantiates a counter, for the `counter` variable of this example, that

	FV	CPU	BDD	FV	CPU	BDD	FV	CPU	BDD	FV	CPU	BDD	FV	CPU	BDD
bbsse	3	0.9	0.65	4	0.72	0.48	5	0.67	0.33	6	0.66	0.27	7	0.91	0.33
cse	3	0.87	0.34	4	0.82	0.23	5	1.33	0.16	6	1.1	0.13	7	0.89	0.15
dk14	3	0.24	0.21	4	0.47	0.18	5	0.54	0.15	6	0.5	0.12	7	0.54	0.14
dk16	3	0.89	0.19	4	0.96	0.12	5	1.04	0.11	6	0.99	0.08	7	0.9	0.08
dk27	3	0.63	0.96	4	0.54	0.74	5	0.75	1.02	6	0.84	1.19	7	0.89	1.18
dk512	3	0.36	0.17	4	0.3	0.12	5	0.24	0.1	6	0.38	0.13	7	0.47	0.17
ex1	3	1.23	0.54	4	1.01	0.44	5	0.43	0.31	6	0.55	0.3	7	0.97	0.23
ex2	3	0.53	0.52	4	0.36	0.4	5	0.46	0.35	6	0.49	0.36	7	0.45	0.35
ex5	3	0.98	1.82	4	0.87	1.33	5	1.02	1.49	6	0.82	1.47	7	0.82	1.57
ex6	3	1.05	0.67	4	1.09	0.55	5	1.04	0.46	6	0.43	0.45	7	0.51	0.55
f_master	3	0.44	0.16	4	0.28	0.11	5	0.36	0.1	6	0.42	0.08	7	0.39	0.08
f_math	3	0.65	0.26	4	0.75	0.4	5	0.69	0.12	6	0.74	0.11	7	0.46	0.07
f_sync	3	0.38	0.43	4	0.41	0.28	5	0.5	0.27	6	0.43	0.36	7	0.44	0.46
opus	3	0.51	0.87	4	0.52	0.7	5	0.49	0.54	6	0.65	0.51	7	0.63	0.46
planet	3	0.25	0.26	4	0.15	0.16	5	0.75	0.07	6	0.91	0.05	7	1.09	0.05
pma	3	0.26	0.27	4	0.33	0.07	5	0.52	0.08	6	0.64	0.07	7	0.97	0.07
s1	3	0.47	0.14	4	0.61	0.16	5	0.22	0.2	6	1.02	0.12	7	1.27	0.1
s1488	3	0.78	0.18	4	0.33	0.17	5	0.31	0.1	6	0.29	0.1	7	0.32	0.09
s1494	3	0.27	0.19	4	0.3	0.2	5	0.3	0.16	6	0.25	0.13	7	0.36	0.1
s27	3	0.5	0.4	4	0.46	0.54	5	0.38	0.59	6	0.54	0.54	7	0.46	0.55
s298	3	0.18	0.21	4	0.11	0.11	5	0.29	0.08	6	0.11	0.08	7	0.14	0.05
s510	3	0.24	0.29	4	0.87	0.11	5	0.86	0.09	6	1.21	0.08	7	1.31	0.08
% sav.		42.7	55.8		**44.3**	65.5		40.0	68.7		36.5	**69.4**		31.0	68.6

Table 2: Effectiveness of faults clustering.

is able to perform a transition from state **MAX-1** to state **MAX** even if the so specified VHDL description does not allow this transition. Thus, it is extremely likely that this description will produce *sequentially-non-excitable* faults.

The adopted functional fault model considers the previously described situations by distinguishing between *explicit* and *implicit* states. The translation of VHDL to BDDs identifies also a set of explicit states and the set of unused state codes, thus partitioning present-state variables (s) into:

- explicit state variables, s_e,
- implicit state variables, s_i.

A functional fault is not considered if the corresponding faulty function $R_F(x, z, s, t, f)$ differs from the fault-free function for transitions outgoing from explicit unused state codes only. This is equivalent to avoiding the consideration of stuck-at faults on gates which are not included into the synthesized gate-level description of the device. On the contrary, functional faults affecting transitions outgoing from unreachable implicit state variables (s_i) are considered since commercial RTL and logic synthesis tools are not able to identify unreachable implicit states. Untested faults activated from such states are likely included into the synthesized gate-level description thus decreasing the testability of such circuits. In conclusion, the proposed functional fault model is able to identify potentially untestable faults before the actual synthesis.

4 Experimental Results

The proposed testability analysis has been implemented by using the commercial tool LEDA LVS for VHDL parsing, the CUDD binary decision diagrams library [14] for BDD manipulation and the Berkeley SIS [13] environment for interface and utilities. The current implementation is composed of more than 120K C code lines. Preliminary experiments have been carried out on academic controllers (MCNC benchmarks) and on industrial VHDL circuits.

Fist of all we investigated the impact of the fault clustering strategy on test pattern generation. Results are reported in Table 2. Two performance indices are reported for increasing cluster size (FV, added faulty variables): the total CPU time and total memory occupation (# BDD nodes). Such values are normalized to the application of the TPG algorithm to a cluster composed by an unique fault. The concurrent analysis of faults is an attractive strategy since it sensibly decreases the required CPU time up to more than 40% on average. Moreover, the total memory occupation decreases, since the same BDD structures are shared between faults. Moreover, it is possible to observe that there is a limit in the number of faults included in a cluster after that the TPG performance begins to decrease. For this class of circuits this limit seems to be between 4 and 6 faulty variables (FV), that is between 16 and 64 concurrently analyzed faults.

Moreover, the gate-level fault simulation of functional test sequences generates, in all examined cases, the *full* stuck-at fault coverage. Such a result is really promising.

name	VHDL Lines	#In	#Out	#F.F.	#Gates
TX	335	10	10	16	210

Table 3: VHDL description examined

About testability measure, we examined an industrial controller part of a telcom device. Its characteristics are reported in Table 3 as number of VHDL instructions, input and output bits, flip-flops (#F.F.) and gates (#Gates). The total number of flip-flops is partitioned into 6 flip-flops used to represent the actual state of the controller and 10 flip-flops representing implicit states. The gate-level implementation of

the circuit has been obtained by using Mentor Autologic.

Results reported in Table 4 compare the testability measure obtained by applying a structural fault model (*Structural*) and two different functional fault models. Fault coverage (#F.C.) for the Structural fault model is expressed as number of tested stuck-at faults, while fault coverage for the functional fault models is expressed as number of tested errors. The second functional fault model (*Functional-Exp.*) differs from the first one (*Functional*) since it considers all criteria reported in Section 3.3.

fault-model	#Faults	#Untested Faults	#F.C.
Functional	487	95	80.5%
Functional-Exp.	413	21	94.9%
Structural	502	74	95.6%

Table 4: Effectiveness of the testability measure

The relevant results concern the testability estimation obtained with the second functional fault model. This measure is higher than the estimation obtained by using the first functional fault model, and it is really close to the structural testability level. Note also that the simulation of functional test sequences on the gate-level representation of the circuit reaches the same fault coverage (95.6%) achieved by directly generating test sequences for stuck-at faults. On the contrary, random test generation applied at the gate level does not cover more than 65% of stuck-at faults.

5 Concluding Remarks

This paper has described a methodology for testability estimation of VHDL descriptions of sequential controllers. The overall methodology is based on binary decision diagrams that allow the analysis of complex descriptions and extend methodologies based on manipulation of state transition graphs. The deterministic sequential test generation algorithm is able to analyze large set of faults by injecting groups of single faults in the BDD based description. As show in the experimental results section fault clustering sensibly reduces analysis time. Moreover, the fault model presented in [8] is here improved by taking into account faults on non-essential primes, faults on shared logic and on *don't care functionalities*. Such a further classification allows high correlation between testability of RT and logic level descriptions. Therefore, testability problems can be eventually identified and removed before synthesis. Experimental results have shown the effectiveness of the fault clustering approach and the high correlation between testability estimation at the functional and logic level.

Future work concerns the further improvement of the proposed fault model and of the BDD based algorithms. Moreover, we will apply our approach to a large set of industrial examples in order to further evaluate the effectiveness of this approach.

References

[1] M. Abramovici, A.D. Breuer, and A.D. Friedman. Digital systems testing and testable design. *Computer Science Press*, 1990.

[2] C. Bolchini, G. Buonanno, F. Ferrandi, F. Fummi, D. Sciuto, M. Bombana, P. Cavalloro, and PM. Borrego. Definition of methodology for testability analysis at the RTL and CDFG levels requirement specs for functional pattern quality evaluator. *Technical Report of Deliverable 2.3.A, Esprit project n.20616 - REQUEST*, 1996.

[3] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):79–85, August 1986.

[4] G. Buonanno, F. Fummi, D. Sciuto, and F. Lombardi. FsmTest: Functional test generation for sequential circuits. *INTEGRATION: the VLSI Journal*, 20:303–325, 1996.

[5] K.T. Cheng and J.Y. Jou. A single-state-transition fault model for sequential machines. *Proc. IEEE ICCAD*, pages 226–229, 1990.

[6] H. Cho, G.D. Hachtel, and F. Somenzi. Redundancy identification/removal and test generation for sequential circuits using implicit state enumeration. *IEEE Trans. on Computers*, 12(7):935–945, July 1993.

[7] O. Coudert and J.C. Madre. Implicit and incremental computation of primes and essential primes of boolean functions. *Proc. ACM/IEEE DAC*, pages 36–39, 1992.

[8] F. Ferrandi, F. Fummi, E. Macii, M. Poncino, and D. Sciuto. BDD-based testability estimation of VHDL designs. *Proc. EuroDAC/EuroVHDL*, pages 444–449, 1996.

[9] G.D. Hachtel and F. Somenzi. Logic synthesis and verification algorithms. *Kluwer Academic Publishers*, 1996.

[10] S.I. Minato. Zero-suppressed BDDs for set manipulation in combinational problems. *Proc. ACM/IEEE DAC*, 1988.

[11] V. Pla, J.F. Santucci, and N. Giambiasi. On the modelling and testing of vhdl behavioral descriptions of sequential circuits. *Proc. EuroVHDL*, pages 228–235, 1993.

[12] I. Pomeranz and S.M. Reddy. On achieving a complete fault coverage for sequential machines. *IEEE Trans. on CAD/ICAS*, 13(3):378–386, March 1994.

[13] SIS: A system for sequential circuit synthesis. *Electronics Research Lab. Mem. UCB/ERL M92/41 University of California, Berkeley*, 1992.

[14] F. Somenzi. CUDD: CU decision diagram package, version 2.1.2. *Department of Electrical and Computer Engineering, University of Colorado at Boulder*, 1997.

[15] M.K. Srinivas, J. Jacob, and V.D. Agrawal. Functional test generation for synchronous sequential circuits. *IEEE Trans. on CAD/ICAS*, 15(7):831–843, 1996.

[16] Z. Peng X. Gu, K. Kuchcinski. Testability analysis and improvement from VHDL behavioral specifications. *Proc. EuroVHDL*, pages 644–649, 1994.