Early Prediction of Testability by Analyzing Behavioral VHDL Specifications

Erik Larsson and Zebo Peng
Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden
Phone: +46 13 284046, Fax: +46 13 282666
Email: {erila, zpe}@ida.liu.se

Abstract

A behavioral testability analysis technique is proposed for early prediction of testability by analyzing behavioral specifications written in VHDL. The technique extracts functional properties by an analysis of variable range, statement hardness and dependency relation between variables of the specification. It predicts the testability for the whole design with a low computational cost. Another feature of the proposed technique is that it reduces the number of design iterations since the hard-to-test parts of the design are grouped and several testability improvement techniques can be applied in each design iteration. Experimental results show that the behavioral testability analysis technique predicts the hard-to-test parts accurately and efficiently.

1. Introduction

Due to the increasing complexity of digital designs, modeling techniques at the behavioral level have been developed and many design and verification activities take place at this level. In order to reduce the test generation and application complexity, it is also important to predict testability of a design at the behavioral level and to make sure the final design is testable. At the behavioral level the functional properties of the design is explicitly captured and can be used to speed up testability analysis. These informations are difficult to extract from a gate-level design.

Early work in testability analysis is, however, usually carried out at the gate level [1,2,3,4,5,6]. A metrics for testability at the gate level should, in general, show that a line close to a primary input is easily controlled and a line close to a primary output is easily observable. Rutman proposed such a metrics based on logical depth to compute the controllability and the observability for the lines in a circuit [7]. Using Rutman's formulas, Chen *et. al*. introduced a technique to perform sensitivity analysis [8]. The basic idea behind it is that an introduction of a Design for Testability (DFT) technique should affect the global testability and not only the place where the DFT improvement has been applied. The main draw-back, however, is that the computational complexity is so high that the approach is not usable in practice for large circuits [9].

On the register-transfer level a common approach to testability analysis has been based on the probabilities of data [10,11]. Each operation reduces the probability of controlling/observing a line deep in the design. Gu *et. al.* proposed such a testability metrics considering only the data path [12]. A similar testability metrics is proposed by Flottes *et. al*., where the controller is assumed to be fully testable [13]. However, a separately testable controller and a separately testable data path do not imply that the combination is easily tested [14].

On the behavioral level Chen *et. al.* introduced a path analysis method to classify, based on controllabilities, the variables into two classes, completely controllable (CC) and noncompletely controllable (NCC) [15]. The classification is based on the ability to control the bits in the variables. If all bits are controllable the variable is classified as CC otherwise as NCC. The few classes and the strict rules for a variable to be classified as a CC variable make most variables to be classified as NNC variables [16]. The Effectiveness measure evaluates the relative improvement for inserting a DFT technique for each NCC variable, which is similar to the sensitivity analysis approach.

A common weakness with the above proposed testability analysis techniques is the way feed-back loops in the design are handled. Loops are often the cause of problems in test generation and must be considered. However, loops also cause problems for most testability analysis approaches. Flottes *et. al.* [13] and Chen *et. al.* [15] assume therefore that no loops exist and, Gu *et. al.* [12] have only a simple heuristics for loop estimation.

Another common draw-back with the proposed testability analysis techniques is that the heuristics for selecting hard-to-test parts select only one part in each design iteration. Therefore after each analysis, only one hard-to-test part is improved and then the design is re-analyzed and its testability is improved again if necessary.

This approach is usually justified by that it keeps the overhead introduced by DFT techniques at a minimum and the introduction of a DFT technique improves usually not only the place where it is applied but also the global testability. However, for large designs the number of iterations will also be very large, resulting in a long design time. A way to keep the DFT overhead small and reduce the number of design iterations is to group the hard-to-test parts according to how they depend on each other. In each iteration, hard-to-test parts from different groups can then be selected and improved without affecting each other.

In this paper we propose a behavioral testability analysis technique which predicts the hard-to-test parts by extracting functional properties from a behavioral VHDL specification. Our technique analyzes the testability of the whole design with a low computational cost and it is based on analysis of variable range, statement hardness and dependency relation between variables. The variable range analysis capture the value-range for a certain variable at a certain line in the behavioral specification. The statement hardness captures the testability hardness which different statements lead to when implemented in hardware. Variables are also grouped according to how they depend on each other to make it possible to apply several DFT techniques in each design iteration and still keep the DFT overhead as small as possible.

The rest of the paper is organized as follows. First preliminaries are given in section 2. In section 3, the testability metrics is presented and the application of the behavioral testability analysis technique for partial scan selection is described in section 4. The analysis algorithm is given in section 5 and in section 6 the variable dependency is explained.. The experimental results are given in section 7 and conclusions are in section 8.

2. Preliminaries

In this section we first discuss the constraints on the behavioral VHDL specification and then give an introduction to the used test strategy.

VHDL is a hardware description language which can be used to model a design at various abstraction levels. At the behavioral level a subset of the VHDL language can be used as an input to a high-level synthesis tool. We assume that the behavioral specification is a synthesizable subset of behavioral VHDL [17].

For our testability definitions we assume that Automatic Test Pattern Generation (ATPG) is used and it is random based and/or deterministic based and oriented towards the commonly used stuck-at fault model. Many ATPG-tools use randomly generated test vectors for finding many easily detected faults and then deterministically generated vectors for harder faults.

To ease test vector generation the technique of synthesis for testability has been widely used. It contains synthesis as well as post-synthesis transformations. A technique of the latter type is the scan technique which increases the controllability and observability for hard-to-test flip-flops [12]. Our testability analysis technique can be used to guide the selection of scan registers in a partial scan based method, as will be discussed in section 4.

3. Behavioral Testability Metrics

Our behavioral testability metrics consists of *Variable Range* and *Statement Hardness*. The *Variable Range, VR,* captures the value range for a variable at a certain line in the behavioral specification, while the *Statement Hardness, SH,* captures the testability hardness which different statements lead to when implemented in hardware.

3.1 Variable Range

If the value range of a variable is reduced at a line in the behavioral specification, it tend to form a hard to test part.

Definition 1: $VR(l, v)$ gives the variable range of variable v at line l in the behavioral specification where $l \in L$. L is the set of lines in the behavioral specification.

For example, if a variable v can have values in the range [-10:10] and [15:20] at line l then $VR(l, v)$ = [-10:10, 15:20].

Definition 2: $defVR(v)$ is the defined variable range for a variable v.

Typically $defVR(v)$ equals the full range of values defined for a variable v. For example, $defVR(v)$ for a 16-bit register declared as a positive integer is $[0:2^{16}-1]$.

Definition 3: Let the *relative variable range*, *RVR*, for a variable v at line l, where $l \in L$, be:

$$RVR(l, v) = \frac{|VR(l, v)|}{|defVR(v)|}$$

Statements	Variable A	RVR(l,A)
A:=A_IN;	0..31	1
IF A<10 THEN	0..9	0.31
A:=A+5;	5..14	0.31
END IF	-	-
A_OUT<=A;	0..31	1

Figure 1.An example of Relative Variable Range Analysis.

op	area
+	166
-	192
*	1527
/	3855
grt,less, neq, lee, gre	127
eql	65
:=, <=	131
mux	95
if	condition
for	434
while	condition+434

Figure 2.The estimated area for 16 bits' operations.

```
ENTITY ex1 IS
  PORT( A_IN, B_IN : IN INTEGER;
        C_OUT : OUT INTEGER );
END;
ARCHITECTURE behave OF ex1 IS
  VARIABLE A, B, C: INTEGER;
BEGIN
  PROCESS BEGIN    l   A    C    SH   RSH
    A := A_IN;     1   131  1    131  0.66
    B := B_IN;     2   131  1    131  0.66
    IF A < 10 THEN 3   127  3    381  1.91
      A := A - 5;  4   323  1    323  1.62
      C := B + A;  5   297  1    297  1.49
    END IF         6   0    1    0    0
    C_OUT <= C;    7   131  1    131  0.66
  END PROCESS;
END;
```

Figure 3. An example of Statement Hardness Analysis.
A is the area, C is the control and l is the line.

The relative variable range of a behavioral VHDL example is shown in figure 1, where `A_IN` is an input port, `A_OUT` is an output port and `A` is defined as: `VARIABLE A: INTEGER RANGE 0 TO 31`.

3.2 Statement Hardness

Testing a module depend on the complexity of the module. Testing complex modules is harder compared with testing simple modules.

We assume that the estimated *area*(*op, w*) for the operation *op* with word length *w* is stored in a module library. A module library for 16 bits operations made of 0.8 µm AMS technology can be found in figure 2.

For instance, an adder has the area of 166 *mils*^{2}. For the `FOR`-statement we can estimate the area, but for statements such as `IF` and `WHILE` the estimation will also depend on the condition.

Definition 4: Let *control*(*l*) of a line *l* denote the number of statements controlled by line *l*, including itself.

For example, the statement: `A:=A-5` at line 4 in figure 3 do not control any other statements, i.e *control*(*4*) = *1*. The statement `IF A<10 THEN` at line 3, on the other hand, controls 3 statements (line 3, 4 and 5) and *control*(*3*)=*3*.

Definition 5: Let the *Statement Hardness*, *SH(l)*, at a line *l* in the behavioral specification be:

$$SH(l) = \sum_{op \in OP} area(op, w) \times control(l)$$

where $l \in L$, and *OP* is the set of operations at line *l* in the behavioral specification.

Definition 6: Let the relative statement hardness, *RSH(l)*, at line *l* in the behavioral description, $l \in L$, be:

$$RSH(l) = \frac{SH(l)}{\sum_{l \in L} SH(l)/L}$$

For example the statement hardness is calculated on the VHDL code in figure 3, using the library in figure 2.

4. Application of the Behavioral Testability Metrics

The behavioral testability metrics is not targeted at any particular DFT improvement technique. In this section we will show how the Behavioral Testability Metrics can be applied when partial scan is used.

The variables in the behavioral VHDL will be implemented as registers on structural level and as flip-flops at gate level. In high-level synthesis the variables are mapped to registers. Several techniques have been developed to minimize the number of registers. However, here we assume that each variable is implemented as a dedicated register.

The flip-flops in the controller, the *state variable*, can not be derived from any variable in the behavioral specification. However, the flip-flops in the state variable are often the cause of problems in test generation and test application and must be considered.

The state variable is indirectly used at every line in the behavioral specification and it can only have one value at each state. If we assume that each line will corresponds to one state then the value range at each line is $1/L$. We can then use definition 3 to get its *RVR* and for the statement hardness we use definition 6. Let the state variable be included in the set of variables.

Definition 7: The testability $T(v)$ for a variable v is given by:

$$T(v) = \sum_{l \in L} \frac{n}{m} \times (RSH(l) + (1 - RVR(l, v)) \times L)$$

where n is the number of times the variable v occur at line l and m is the total number of lines where variable v occur.

Since any value can be stored and observed in a scanned register we let the value range be equals to *defVR(v)* when a register/variable is scanned.

Definition 8: A scanned variable v has the *Value Range, VR(l,v)*, equals *defVR*(v).

From definition 3 and 8 and we can conclude that the *Relative Value Range* for a scanned variable is 1.

Definition 9: Scanning a variable v reduces the Statement Hardness, $SH(l)$, to:

$$SH(l) = \frac{SH(l) \cdot (i - j)}{i}$$

where i is the number of variables and j the number of scanned variables at line l. Note that the state variable is affecting every line and that we include it in the number of variables.

The statement hardness is reduced when a statement including a scanned variable is scanned. For example, with a statement `A:=B+C` where `B` is scanned, we reduce the statement hardness for all variables at the line by a fourth (3 variables + state variable).

Scanning a variable `A` at a line such as `A:=B+5` improves the testability for variable `A` and the testability for variable `B` as well as the state variable.

Definition 10: Scanning variable v at line l makes $control(l)=1$.

For example, if the state variable is scanned, $control(l)=1$ for all l since any state can easily be reached.

5. Behavioral Testability Analysis Algorithm

The behavioral testability analysis algorithm calculates the testability metrics, value range and statement hardness for all lines in the behavioral specification.

Definition 11: Variable v at a line l is a *target variable* if v is the left-hand variable in an assignment or if v is used and restricted in an `IF`, `WHILE` or `FOR` statement.

For instance, in a statement: `A:=B+C` variable `A` is a target variable. Variable `A` is also a target variable in a statement: `IF A<10 THEN`.

The pseudo-code for the algorithm is given in figure 4, where L is the set of lines in the behavioral specification, V is the set of variables, $OP(l)$ is the set of operations at a line l. The function get_area reads the area from the module library.

The algorithm consists mainly of traversals over the number of lines, L, in the behavioral specification and the variables, V. The algorithm has the complexity of $O(2L^V)$.

```
SH= 0;
FOR l =  1 TO |L| DO {
   area = 0; control = 1;
   FOR op = 1 TO |OP(l)| DO  {  area = area + get_area( op ); }
   IF l IN {"IF","FOR","WHILE"} THEN  {
       k = l;
       WHILE  NOT  k IN {"END} DO  { control = control + 1; k = k + 1; }}
   SH = SH + area * control;
   FOR i = 1 TO |V| {
       IF target_variable( vi, l) THEN  {  RVR(vi) = | VR(l,v) | / | defVR(v) |; }
       IF ni,l <> 0 THEN { mi = mi + 1; }  // mi = # lines where vi is used,
   }}                                       // ni,l = # times vi is used in l;
FOR l =  1 TO |L| DO  {
   RSH(l) = SH(l) * L / SH;
   FOR i=1 TO |V| DO  {  T(vi) = T(vi) + ni,l/mi*(RSH(l)+(1-RVR(l,v))*L); }}
```

Figure 4.The testability analysis algorithm.

6. Variable Dependency

The number of design iterations can be reduced if more than one hard-to-test part can be selected and improved by a DFT technique in each iteration. The aim of variable dependency is to group the variables based on their dependency.

The testing problem is mainly a problem of justifying values from a primary input to a variable and then propagating values from a variable to a primary output. The controllable paths for a variable v is used to justify a value for variable v and the observable paths are used to propagate the value of the variable to a primary output.

Any path from a primary input to a variable can be used for the justification process, and any path from a variable to a primary output can be used for the propagation process. However, we assume that the justification process will mainly use the shortest path from some primary input to the and the propagating process the shortest path from the variable to a primary output.

Let **G(V, E)** be a directed graph where a vertex v in **V** corresponds to a variable in the behavioral specification (the state variable is not considered).

Definition 12: A *start vertex* is a vertex which gets its value from a primary input and an *end vertex* is either a variable which is connected to a primary output or a variable which is not used later.

An example of an end vertex is the index variable in a FOR-statement.

Definition 13: For a statement of the form $v_i = v_j \; op \; v_k$, where v_i, v_j and v_k are variables and *op* is an operation, there exists an edge (v_j, v_i) in **E** from vertex v_j to v_i and an edge (v_k, v_i) in **E** from vertex v_k to v_i.

Definition 14: A path, P_i, is a sequence of edges $\{(v_o, v_1), (v_1, v_2),..., (v_{n-1}, v_n)\}$ where v_0 is a start vertex and v_n is an end vertex.

Definition 15: The *Shortest Controllable Path*, *SCP*(v), for a variable v, is the shortest path from a start vertex to variable v, and the *Shortest Observable Paths*, *SOP*(v), is the shortest path from variable v to a primary output.

The SCP and SOP information will be used to group variables together, in such a way that only one variable will be selected for testability improvement from a group during a design iteration.

An algorithm for grouping variables is given in Pseudo-code in figure 5. First the testabilities for all variables are ranged so that T_1 is the hardest to test and T_V is the easiest to test (V is the total number of variables). The hardest variable, T_1, is always selected for DFT improvement. Then the algorithm checks so that variables that use parts of the same shortest observable path and parts of the same shortest controllable path are grouped together.

Sort the testabilities $T(v)$ so that T_1 is the hardest and T_V is the easiest to test.
Select T_1 for DFT improvement.
FOR i = 2 TO |V|
    Diff = $\varnothing$;
    FOR j = 1 TO i - 1
        Diff = Diff + $SOP(v_i) \cap SOP(v_j)$ + $SCP(v_i) \cap SCP(v_j)$;
    IF DIFF = $\varnothing$ THEN select T_i for DFT improvement

Figure 5.Algorithm for grouping variables.

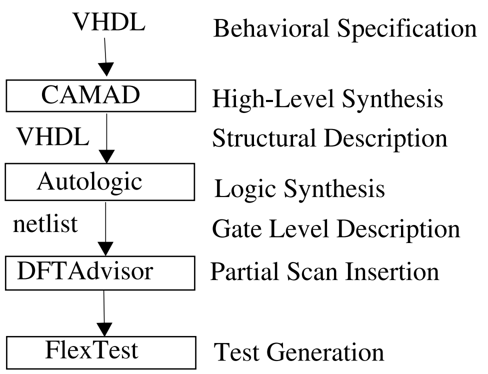

Figure 6. The design flow.

```
ENTITY diff IS
  PORT(x_in,y_in,u_in: IN INTEGER;
      x_out,y_out,u_out: OUT INTEGER);
END;
ARCHITECTURE behavior OF diff IS
BEGIN
  PROCESS
    CONST          a = 10000;
                   dx = 1;
    VARIABLE x,y,u,x1,y1,u1 : INTEGER;
  BEGIN
    x:=x_in; y:=y_in;u:=u_in;    (1,2,3)
    WHILE x<a LOOP               (4)
      x1:=x+dx;                  (5)
      u1:=u-(3*x)*(u*dx)-3*y*dx; (6)
      y1:=y+u*dx;                (7)
      x:=x1;y:=y1;u:=u1;         (8,9,10)
    END LOOP;                    (11)
    xout<=x;yout<=y;uout<=u;     (12,13,14)
  END PROCESS;
END behavior;
```

Figure 7. The behavioral description of the Diff benchmark.

7. Experimental Results

In this section we present our experimental results. We have performed a detailed analysis on the Differential Equation benchmark, Diff. We have also compared the efficiency of the testability prediction on the behavioral level with those of the gate-level, using a set of benchmarks.

We have used the CAMAD high-level synthesis tool [18] and Mentor Graphics logic synthesis and test generation tools as the experimental platform. The behavioral VHDL specification was given as input to the CAMAD high-level synthesis tool, which produced a structural VHDL description. Logic synthesis of the structural description was performed by Autologic and a netlist was generated. For partial scan insertion we used DFTAdvisor and for test vector generation we used FlexTest, see figure 6. The benchmarks we have used are Diff [19], Sqrt [20] Mag[20], Dct [21] and Tseng [22].

The behavioral specification for the Diff benchmark is given in behavioral VHDL, in figure 7. The variable range analysis, the statement hardness analysis and the combination of them into one testability metrics, table 1.

After the analysis of the Diff benchmark we made experiments to see if the ranging of variables was correct, the results are given in table 3. In the table the state register is ranged as the hardest and by scanning it the fault coverage is 97.28%. On the other hand, register y1, is considered to be not difficult to be tested, therefore scanning it will not improve the design’s testability, we get only 15.61% fault coverage as the result.

The comparison between analysis at behavioral and gate level was performed as follows. First we did a testability analysis on the behavioral VHDL to find hard-to-test variables and rank them. The variables which are hardest to test are scanned. Secondly, we let the Mentor Graphics gate level DFT-tool select the same number of flip-flops as was selected by our behavioral analysis. The results are shown in table 2 where the fault coverage is almost the same for all benchmarks except for the Diff benchmark where the behavioral analysis outperformed the gate-level analysis.

The behavioral testability analysis determined the number of flip-flops to scan. However, the gate level tool have the advantage of selecting individual flip-flops from any register. The main advantage of the behavioral testability analysis is that it takes much less time to perform the testability analysis task, and therefore speeds up the design process.

Line	Relative Variable Range							Relative Statement Hardness				Testability						
	x	y	u	x1	y1	u1	state var	area	ctrl.	SH	RSH	x	y	u	x1	y1	u1	state var
1	1	-	-	-	-	-	0.07	131	1	131	0.13	0.13	-	-	-	-	-	13.14
2	-	1	-	-	-	-	0.07	131	1	131	0.13	-	0.13	-	-	-	-	13.14
3	-	-	1	-	-	-	0.07	131	1	131	0.13	-	-	0.13	-	-	-	13.14
4	0.15	-	-	-	-	-	0.07	561	5	2805	2.75	14.6	-	-	-	-	-	15.37
5	0.15	-	-	0.15	-	-	0.07	297	1	297	0.29	12.2	-	-	12.2	-	-	13.31
6	0.15	1	1	-	-	1	0.07	8150	1	8150	8.00	20.0	8.0	16.0	-	-	8.0	21.33
7	-	1	1	-	1	-	0.07	1824	1	1824	1.79	-	1.79	1.79	-	1.79	-	14.76
8	0.15	-	-	0.15	-	-	0.07	131	1	131	0.13	12.0	-	-	12.0	-	-	13.14
9	-	1	-	-	1	-	0.07	131	1	131	0.13	-	0.13	-	-	0.13	-	13.14
10	-	-	1	-	-	1	0.07	131	1	131	0.13	-	-	0.13	-	-	0.13	13.14
11	-	-	-	-	-	-	0.07	0	1	0	0	-	-	-	-	-	-	13.0
12	0.85	-	-	-	-	-	0.07	131	1	131	0.13	2.2	-	-	-	-	-	13.14
13	-	1	-	-	-	-	0.07	131	1	131	0.13	-	0.13	-	-	-	-	13.14
14	-	-	1	-	-	-	0.07	131	1	131	0.13	-	-	0.13	-	-	-	13.14
T(v)												10.2	2.0	3.6	12.1	1.0	4.1	14.0

Table 1: The relative variable range and the statement hardness combined for the Diff benchmark.

Scanned variable	Hardness	Fault coverage (%)
state register	14.0	97.28%
x1	12.1	97.61%
x	10.2	97.69%
u1	4.1	91.78%
u	3.6	92.33%
y	2.0	16.15%
y1	1.0	15.61%
no scan	not appl.	16.44%

Table 3: Experimental results for the Diff benchmark.

Benchmark	# of flipflops	# of scanned flip-flops	Fault coverage(%)	
			Behavioral level analysis	Gate level analysis
Diff	100	4	97.3	13.8
Sqrt	86	70	99.0	99.0
Mag	52	36	95.9	94.9
Dct	164	4	98.8	99.1
Tseng	213	4	96.3	93.9

Table 2: Fault coverage comparison of testability analysis at the behavioral and the gate levels.

8. Conclusions

In this paper we have proposed a behavioral testability analysis technique for early prediction of testability by analyzing the behavioral VHDL specification. The technique is based on analysis of variable range, statement hardness and dependency relations between variables. The testability for the whole design is predicted at a low computational cost.

We have proposed a variable grouping scheme, based on the dependency of variables, which can be used to reduce the number of design iterations since several variables can be improved in each iteration.

By experiments we have shown that the behavioral testability analysis technique can predict the hard-to-test parts accurately and efficient and that the testability analysis results can be used to guide the selection of partial scan registers.

9. References

[1] L. H. Goldstein, Controllability/Observability Analysis of Digital Circuits, *IEEE Trans. on Circuits and Systems*, vol CAS-26, no. 9, Sep. 1979, pp 685-693.

[2] L. H. Goldstein, SCOAP: Sandia Controllability/Observability Analysis Program, *Proceedings of Design Automation Conference*, Proceedings of DAC, June 1980, pp 190-196.

[3] R. Gupta, R. Gupta, M. A. Breuer, The BALLAST Methodology for Structured Partial Scan Design, *IEEE Trans. on Computers,* vol. 39, no. 4, April 1990, pp 538-544.

[4] P. S. Parikh, M. Abramovici, A Cost-Based Approach to Partial Scan, *Proc. of DAC'93.* pp 255-259.

[5] P. S. Parikh, M. Abramovici, Testability-Based Partial Scan Analysis, *J. of Electronic Tesnd appl.,* 7, 61-70, 1995.

[6] M. Abramovici, J. J. Kulikowski, R. K. Roy, The Best Flip-Flops to Scan, *Proceedings of ITC'91,* pp 166-173, 1991

[7] R. A. Rutman, Fault detection test generation for sequential logic by heuristic tree search, *IEEE Computer Group Repository,* R72-187, 1972.

[8] T. Chen, M. Breuer, Automatic Design for Testability Via Testability Measures, *IEEE Trans. on Computer-Aided Design, vol. CAD-4, No. 1, January 1985,* pp 3-11, 1985.

[9] M. Abramovici, M. A. Breuer, and A. D. Friedman, Digital Systems Testing and Testable Design, *IEEE Trans. Computer-Aided Design*, IEEE, 1990, ISBN 0-7803-1062-4.

[10] C. Ramachandran, F. J. Kurdahi, Incorporating the Controller Effects During Register Transfer Level Synthesis, *Proc. of the European Design and Test Conf. '94.* pp 308-313

[11] C. H. Chen, P. R. Menon, An Approach to Functional Level Testability Analysis, *Proc. of ITC'89,* pp 373-380. 1989.

[12] X. Gu, K. Kuchcinski and Z. Peng, Testability Analysis and Improvement from VHDL Behavioral Specifications, *Proc. EURO-DAC'94*, pp. 644-649, Grenoble, September 1994.

[13] M. L. Flottes, R. Pires, B. Rouzeyre, Analyzing testability from behavioral to RT level, *Proceeding of European Design and Test Conference,* pp 158-165, March 1997.

[14] S. Dey, V. Gangaram, M. Potkonjak, A Controller-Based Design-for-Testability Technique for Controller-Data Path Circuits, *Proceeding of ICCAD,* pp 640-645, 1995.

[15] C.-H. Chen, T. Karnik and D. G. Saab, Structural and Behavioral Synthesis for Testability Techniques, *IEEE Trans. Computer-Aided Design*, 13(6), pp. 777-85, 1994.

[16] J. Lee, J. H. Patel, Testability Analysis Based on Structural and Behavioral Information, *Proceedings of the IEEE VLSI Test Symposium,* pp 139-145, April 1993.

[17] P. Eles, K Kuchcinski, Z. Peng, M. Minea, Compiling VHDL into a High-Level Synthesis Design Representation, *Proceedings of the EURO-DAC,* 7-10 September 1992, pp 604-609.

[18] Z. Peng, K. Kuchcinski, Automated Transformation of Algorithms into Register-Transfer Level Implementations, *IEEE Trans. on CAD of Int. Circ. & Sys.,* no 2, Feb. 1994, pp 150-166.

[19] P. Paulin, J. Knight, Force-directed scheduling for behavioral synthesis of ASIC's, *IEEE Trans. on Computer-Aided Design,* vol 8, June 1989, pp 661-679.

[20] H. Trickey, Compiling Pascal Programs into Silicon, *Ph. D. thesis,* Dept. of Comp. Science, Stanford University, 1985.

[21] G. Krishnamoorthy, J. A. Nestor, Data Path Allocation using an Extended Binding Model, *Proc. Design Automation Conf.,* pp 279-284, June 1992.

[22] C. Tseng, D. P. Siewiorek, A Procedure for the Automated Synthesis of Bus Style Systems, *Proc. 20th Design Automation Conference*, pp 490-496, 1983.