Validation Vector Grade (VVG): A New Coverage Metric for Validation and Test

*Pradip A. Thaker**
Hughes Network Systems
Germantown, MD 20876 USA
pthaker@hns.com

Vishwani D. Agrawal
Bell Labs, Lucent Technologies
Murray Hill, NJ 07974 USA
va@research.bell-labs.com

Mona E. Zaghloul
George Washington University
Washington, DC 20052 USA
zaghloul@seas.gwu.edu

Abstract

Current code coverage metrics used in high level VLSI design methodology are based on statement, branch, toggle and condition coverages of the HDL code obtained by simulating validation vectors. These measures allow the designer to find sections of the HDL code not executed during validation. Feedback from the code coverage analysis helps generate additional vectors to exercise previously unexercised functionality. In this paper, we explore the relationship between results of RTL code coverage and the gate level fault coverage. Based on the observations of this empirical study we propose a new and improved code coverage metric "Validation Vector Grade (VVG)." The VVG-approach modifies code coverage metrics by adding the concepts of observability and arithmetic fault library. VVG is an improved validation metric, which can also be used for early testability analysis at the RT level. Results of the VVG-approach at RT level are shown to be good indicators of the structural fault coverage. Experiments on actual telecommunication VLSI chip designs show that the VVG reported at the RT level can predict the post-synthesis gate level fault coverage with an error margin less than 4%. Also, the improvements in VVG at the RT level due to high-level design changes or added vectors track improvements in gate level fault coverage.

1. Introduction

Advances in EDA technology have increased the size and complexity of VLSI circuits making it challenging to develop comprehensive validation procedures. HDL-based design approach has reduced the actual design time allowing designers to spend relatively more time on validation. The code coverage analysis technique has been popular in the software arena [13]. With the use of hardware description languages, the code coverage technique of software verification serves as a feedback mechanism for hardware design engineers in developing comprehensive validation suites. Various practical approaches [14] and coverage analysis methods [8, 28] have been proposed in recent years. The commonly used code coverage analysis technique offers five quality metric numbers: statement execution coverage, condition coverage (also known as expression coverage), branch coverage (also known as decision coverage), toggle coverage and path coverage [8]. If any coverage is found to be low, analysis of the unused code helps determine additional tests needed.

We explore the relationship between the HDL code coverage and the testability properties of the resulting structural netlist. The current code coverage technique only emphasizes controllabilities of internal RTL variables from primary inputs while ignoring their observabilities at primary outputs. Also, only indicators of code coverage for arithmetic components are statement and toggle coverages which fail to identify testability aspects due to compact representation of arithmetic components at the RT level. Based on these observations, we modified RTL code coverage metric by adding an observability dimension and better modeling of arithmetic components. We call this new and improved RTL code coverage metric "Validation Vector Grade (VVG)." Experimental results show that validation vector grade is quite close to the fault coverage achieved at the gate level when designs are converted to technology-specific netlists. Thus, VVG offers a common metric for validation and test and allows detection of testability problems at an early stage of a design cycle. The vectors generated for verification can be used for production testing of silicon.

In Section 2, we present results and observations of an experimental study that investigates the correlation of RTL code coverage and the gate level fault coverage. Section 3 describes the new validation vector grade approach. In Section 4, we describe the application of VVG for testability analysis and fault coverage prediction. In Section 5, we describe the limitations (validation point of view) of current RTL code coverage technique and show how VVG approach overcomes them. In Section 6, we conclude the paper by pointing out the usefulness of

*A PhD candidate at George Washington University. This research was conducted for the dissertation.

VVG-approach in the area of validation and test. Appendix A gives a fault injection example. Appendix B provides a list of Verilog constructs currently supported in the VVG analysis.

2. Empirical Study of RTL Code Coverage and Gate Level Fault Coverage

Several circuits that were fully designed at the RT level were used as test cases for our analysis. We developed comprehensive validation suites for all of these designs with feedback obtained from TransEDA's *VeriSure* code coverage analysis tool [27]. *VeriSure* accepts the HDL code and simulation testbench as inputs and generates various types of code coverages, such as statement coverage, branch coverage, basic sub-condition coverage, multiple sub-condition coverage, toggle coverage and path coverage. The simulation testbench used in the HDL environment contains input stimulus to the design. We synthesized selected designs by *Synopsys* synthesis toolset [24] and used simulation vectors developed for validation to obtain functional fault coverage for resulting implementation using Cadence's *Verifault-XL*.

The validation vectors developed with feedback from the code coverage analysis methodology exercise all functional modes but each function may not be exercised with all possible data patterns. While these vectors can be considered comprehensive, they are not exhaustive. The resulting coverage numbers reported in Table 1 are considered sufficient for functional verification of practical designs. For example, 17.93% reported as condition coverage is low but after careful analysis, it was found that all of the functional modes had been exercised. The lower condition coverage results from a combination of poor RTL coding style and limitations of the code coverage metric that is discussed in more details later. The results on two designs follow.

Design D1: This design is a system timing controller chip (16 bidirects, 15 primary inputs and 7 primary outputs) designed using Verilog HDL. The number of lines of Verilog code is approximately 3600. The structural implementation obtained after synthesis contains 750 flip-flops and 12145 combinational gates. This design is largely dominated by state machines and controller type of hardware. Therefore, generating a comprehensive validation suite is a real challenge. Comprehensive sets of test vectors were developed using feedback from code coverage analysis tool. The number of vectors in Test 1 is 3755, Test 2 is 306k and Test 3 is 13k. In Table 1, results of RTL code coverage and gate level fault coverage are presented.

Design D2: This design is a digital signal processor chip (10 bidirects, 34 primary inputs and 51 primary outputs) also designed using Verilog HDL. The number of lines of Verilog HDL code is approximately 1850. The structural implementation obtained after synthesis contains 2600 flip-flops, 31303 combinational gates and several embedded RAMs. This design is largely dominated by datapath components such as adders and multipliers with some control logic. Comprehensive sets of test vectors were developed using feedback from code coverage analysis. The number of vectors in Test 1 is 148k, Test 2 is 19k, Test 3 is 138k and Test 4 is 105k. Behavioral models of embedded RAMs were excluded from code coverage and fault coverage. In Table 2, results of RTL code coverage and gate level fault coverage are presented.

Table 1: Data for design D1

Test	RTL Code Coverage (%)				Gate Level Fault Cov. (%)	
	statem.	branch	condit.	toggle	non-coll.	collapse
1	69.6	59.4	16.4	95.2	88.5	87.8
2	62.2	51.6	12.8	93.9	65.6	65.4
3	62.2	54.2	9.7	81.1	69.4	67.5
1+2	74.1	64.4	17.6	96.5	92.6	92.0
1+3	76.3	68.6	17.1	95.9	92.0	91.8
1+2+3	77.0	69.3	17.9	96.7	94.3	93.9

Table 2: Data for design D2

Test	RTL Code Coverage (%)				Gate Level Fault Cov. (%)	
	statem.	branch	condit.	toggle	non-coll.	collapse
1	91.9	78.9	86.5	78.0	60.3	57.7
2	89.9	72.1	79.1	27.5	18.4	18.0
3	94.1	76.8	85.8	84.1	63.3	61.4
4	91.1	76.9	84.6	74.0	55.4	50.9
1+2	92.1	79.1	88.6	80.1	60.3	57.7
1+2+3	95.3	83.6	90.3	94.8	73.1	70.9

Based on the experimental results, we believe that there is some correlation between code coverage and fault coverage for the control logic dominated designs with good observability of internal nodes to primary outputs. For such designs, vectors that are generated by incremental improvements in code coverage also demonstrate incremental improvements in gate level fault coverage. While the designs that are dominated by data path components and/or have poor observability of internal nodes to primary outputs, such correlation is difficult to derive. In either case, it is not possible to derive a one-to-one mapping between current code coverage metrics and gate level fault coverage. The causes for lack of direct relationship between these two quality metrics are identified and discussed below.

The datapath (arithmetic) components are represented by a very high level of abstraction in HDL code. For example, a 32 by 32 bit multiplier can be represented at the RT level with one line of code that does not have any representation of the internal nodes of a

synthesized gate level multiplier circuit. Such HDL code statements inflate the gate count and fault list after synthesis. The only indicators of code coverage for arithmetic components are statement and toggle coverages. However, these two indicators fail to identify testability aspects of arithmetic components at the RT level due to their compact representation.

Even though the RTL code coverage help identify untested code for simulation purposes, it does not help identify testability weaknesses of a design. This is due to the fact that even though RTL code coverage indicators such as statement coverage, toggle coverage, etc., help enhance validation vectors for better controllability of internal RTL variables, they fail to offer any indication of observability of internal RTL code variables to primary outputs. Therefore, various testability weaknesses present at the RT level such as (1) observability blockage resulting from untestable embedded components, and (2) blockage of observability of arithmetic components due to truncation hardware are not detected by code coverage analysis. The testability analysis of gate level netlists for designs D1 and D2 revealed these weaknesses that were not self-evident during code coverage analysis.

High statement, toggle and branch coverages may be necessary for achieving high fault coverage but are not sufficient. Condition coverage is not a good indicator of quality of vectors since it is flawed by the fact that while calculating condition coverage, current code coverage metric does not exclude impossible conditions that are not required to be exercised under normal functional operation. Thus, we identify the need for further enhancement to the code coverage analysis method to deal with the issues of (1) better modeling of arithmetic components and (2) additional dimension to propagate observability of internal RTL variables to primary outputs, so that it can be used as an early testability analysis tool. We modified current code coverage metric to incorporate these findings and named it "Validation Vector Grade (VVG)". The empirical data suggests one-to-one mapping between VVG and gate level fault coverage. Also, VVG will be shown to be a superior metric compared to the current code coverage metrics for validation as well.

3. Validation Vector Grade Approach

The validation vector grade (VVG) extends the concept of toggle coverage of current code coverage technique by adding observability dimension to it while dropping rest of coverage parameters. VVG requires fan-out of each RTL variable to be treated as a unique variable. Instead of five quality metric numbers used by the current code coverage technique, VVG uses only one metric number to report all information reported by current code coverage technique and more. Under VVG approach an RTL variable is reported covered only if it can be controlled from primary inputs and observed at a primary output using a technique that is similar to fault grading. Fault grading is usually performed on gate level netlist. However, VVG is generated by fault grading RTL variables using a stuck-at fault model and RTL code as a design description. The method of RTL fault grading has been first described by Mao and Gulati in [17]. We have refined their approach in several ways.

The RTL fault grading presented in [17] injects faults only on input and internal variables of the design. Also, one is required to run simulations twice and use the average of results of optimistic (with only single stuck-at fault injected per RTL variable) and pessimistic (with single stuck-at fault injected per fan-out branch of an RTL variable) modes to resolve differences between RTL and gate level fault coverage [17]. This is not an efficient solution. Also, RTL fault grading method in [17] does not offer any solution for fault coverage differences resulting from large arithmetic components. It fails to investigate testability relationship between the RTL description and the structural representation. The error margin reported in tabular form is 5.4%, growing to as much as 10% for data presented in graphs [17].

In VVG-approach, we inject stuck-at faults on input variables, internal RTL variables and primary output variables based on a complex fault injection algorithm developed to handle different combinations of RTL constructs described in Appendix B. VVG-approach also requires fan-out of each RTL variable to be treated as a unique variable by injecting fault at each fan-out branch except the branch that goes to input of an embedded submodule. In other words, a separate fault is injected on each fan-out of a variable when variable is used in multiple statements. We further extend the VVG-approach by adding the concept of arithmetic fault library which accounts for faults that are not present at RT level description of arithmetic components. The arithmetic components are represented at a very high level of abstraction in the RT level code. The gate level structure of an arithmetic component does not have any representation at the RT level since the RT level only consists of inputs and outputs described within one statement along with an arithmetic operator. Under the VVG-approach, we replace this compact arithmetic description with an instantiation of a mixed structural-RT level component containing only bitwise operations. This component not only has inputs and outputs but also some description of internal nodes. We selectively sample faults from this component based on estimation of its gate count with respect to the estimated overall gate count of the module.

For the VVG approach, we used Cadence's gate level

fault simulator *Verifault-XL*. *Verifault-XL* has a unique ability for behavioral fault propagation. With some modifications for RTL fault injection using a C++ parser, we were able to use *Verifault-XL* to generate VVG. For more details on the method of running RTL fault grade using *Verifault-XL*, one may refer to [17] and [2]. An example of stuck-at fault injection is given in Appendix A, and Appendix B lists Verilog RTL constructs supported in the VVG approach.

4. VVG as RTL Testability Indicator and Fault Coverage Predictor

Among the proposed high level testability analysis techniques, some focus on calculating controllability and observability measures based on the circuit structures [12]. Others extract testability properties of a design from behavioral or function level descriptions [1, 5, 4, 6, 21, 26]. Among several high level fault simulation techniques proposed in recent years [3, 7, 11, 15, 18, 19], some are limited to circuits with regular structures used as building blocks. Others [7, 11, 18, 19] do not establish relationship between high level fault coverage and structural fault coverage. Several behavioral fault models have been considered including a testability measure [22], and a "difference fault model" [20]. However, both approaches were analyzed and validated only for small designs. For modern RTL designs that are much more complex, these approaches are less likely to produce similarly encouraging results. An RTL DFT technique, described, by Ghosh *et al.* [10], works by extracting the test control data flow information from data path/controller circuit and using it for justification from system inputs and propagation to system outputs. However, none of these techniques explore relationship between validation at RT level and testing of gate level representation. In [17], Mao *et al.* suggests the use of RTL fault grade technique for early testability analysis but contradicts the claim by stating the assumption about impact of logic synthesis on testability properties of resulting gate level representation. Also, RTL fault grade reported in [17] has error margin ranging from 1.12 to 10%. Thus, both the method and model presented in [17] need improvement. As indicated in Section 3, we have refined the approach presented in [17] in many ways. In this section, we present results of experimental data.

We used several RTL designs along with their validation suites to compare effectiveness of VVG to track gate level fault coverage. Table 3 shows the comparison of VVG and gate level fault coverage. The designs D3, D4, D5 and D6 are design blocks in various DSP and communication ASICs with sizes ranging from 600 to 4500 gates. The error margin found between VVG (enhanced with arithmetic fault library) and gate level fault coverage is lower than that reported in [17]. The maximum error margin found in our experiments is not more than 3.6%. Also, VVG approach does not require one to run simulations more than once. VVG also seems to track improvements in gate level fault coverage with low error margin as indicated in Table 4.

Table 3: VVG versus gate-level fault coverage

Design	VVG (%) with enhanced arithmetic library	Gate level fault coverage (%)	
		non-collapsed	collapsed
D3	96.6	96.3	95.5
D4	87.3	83.7	84.5
D5	81.1	82.7	77.8
D6	82.6	82.9	83.7

Table 4: VVG tracking gate-level fault coverage

Design D4	VVG (%) with enhanced arithmetic library	Gate level fault coverage (%)	
		non-collapsed	collapsed
Test 1	72.2	71.4	72.0
Test 2	84.4	81.3	81.7
Test 3	87.3	83.7	84.5

As established with empirical data [25], the architectural testability properties of a gate level netlist are derived from RTL description and are not impacted by constraint-drive logic synthesis. They are also significant portion of overall testability properties of the design. The VVG-approach can be used for architectural testability property analysis at an early (presynthesis) stage of a design cycle. If reported VVG is less than 100%, it can be due to one of the two reasons: (1) insufficient simulation vectors in the test, or (2) poor controllability and/or observability of internal nodes. In the second case, the designer may modify the architectural design for better testability. Thus, VVG seems to be a good RTL testability indicator and gate level fault coverage predictor.

5. VVG as a New Code Coverage Metric For Improved Validation Strategy

Fallah *et al.* [9] have identified the need for an observability-based code coverage metric for functional verification. Their technique relies on an approximate D-calculus in a graph analysis program, OCCOM, that computes observability of internal variables. Their algorithm of tag propagation focuses on reduction of computing effort at the cost of accuracy. Our method (VVG) focuses on accuracy with slightly higher computing cost (1.5 to 4 times for OCCOM vs. 4 to 7 times for VVG, over HDL simulation.) The VVG is an accurate

Table 5: Data for designs D3, D4, D5 and D6

Design	Current RTL Code Coverage Metrics (%)					New Metric VVG (%)
	statem.	branch	subcondition		toggle	
			basic	multiple		
D3	100.0	100.0	100.0	62.5	96.8	96.6
D4	99.1	97.6	100.0	82.5	92.9	87.3
D5	100.0	100.0	100.0	42.2	100.0	89.2
D6	100.0	100.0	90.0	83.3	91.1	82.6

observability-based code coverage metric and provides an early testability analysis and fault coverage prediction.

As in the case of other code coverage techniques and OCCOM [9], VVG approach only provides feedback about the quality of validation vectors. All approaches still require designer to validate output responses of the design module. We compare the VVG approach with the current code coverage technique:

(1) Since the current code coverage technique reports overall quality of vectors using five coverage parameters, the results are often ambiguous as shown in Table 5. For example, for design D5, statement and toggle coverage can each be 100% with very low multiple sub-condition coverage (42.24%). These numbers do not clearly indicate the amount of effort needed to improve the quality of validation vectors. Only an in-depth analysis can reveal if there is a need for additional validation vectors. Such analysis can also sometimes reveal that conditions not covered are functionally redundant.

Our approach reports a single code coverage parameter called Validation Vector Grade (VVG), representing the quality of validation vectors unambiguously. For the above situation, VVG-approach excludes many redundant conditions reducing the complexity of computation. Note that the results of Table 5 did not use the arithmetic fault library. However, only the coverage for D5, which has large arithmetic blocks, has changed (see Table 3.)

(2) The current code coverage technique consumes memory and imposes simulation run-time overhead in a prohibitively expensive fashion. It tracks various types of coverages during simulation by adding callbacks into RTL code [28]. These callbacks (hooks or probes) count occurrences of particular situations such as transitions, conditions, statement executions etc. throughout the simulation. Since designs are getting larger, the performance overhead on simulators has increased exponentially. Also, the current technique does not provide a method of excluding situations during current run that are reported covered in previous runs.

Since the VVG approach uses stuck-at fault models (supported in *Verifault-XL*) for every RTL variable, it is free from performance penalty of the current approach. As soon as a variable is detected, it is dropped from the list of variables. Therefore, as simulation progresses, the performance overhead decreases. However, we do provide a method to keep variables on the active list after they are covered if the designer chooses to do so. Also, one may use random sampling for early estimation of vector quality during validation effort.

(3) Even though a block of code is fully exercised, the validation effort may be meaningless unless the results of code execution propagate through the design. The current code coverage cannot report such problems.

Since VVG approach tracks observability of all internal RTL variables to primary outputs, the problem mentioned above can be detected effortlessly.

(4) As designs get larger and complex, self-checking validation environments are gaining popularity. Validation suites include both input stimuli and the expected output responses. Self-checking validation environments monitor outputs or selected internal nodes during the simulation and report any deviation from the expected response. The current code coverage technique reports how many times an RTL variable toggled without assessing its impact on the primary outputs or selected internal nodes.

In the VVG approach, a variable is reported covered only if it can be controlled from primary inputs and its logic transition impacts primary outputs or the selected internal node (strobe probe). It thus provides an improved self-checking validation environment to detect design bugs.

(5) The current code coverage technique does not give a proper indication of whether the logic is being functionally used during simulation. One can write meaningless or poor tests and still achieve a very high coverage. For example, an enable signal of a counter will be reported covered even if it toggled while counter is in reset state due to active reset signal.

Since VVG-approach draws a parallel from gate level fault simulation, it checks for functional validity of a test to a certain degree. In the above example of a counter, an enable signal is reported covered only if it is properly exercised in a functional manner. If enable signal toggled while counter is in reset state, it will not be covered since it can not be observed at counter outputs.

6. Conclusion

We have investigated the relationship of RTL code coverage reported by the current code coverage technique and gate level fault coverage. We improve the current RTL code coverage metric by adding concepts of observability and arithmetic fault library. This new RTL code coverage metric, VVG, tracks and predicts gate level fault coverage within an acceptable ($< 4\%$)

error margin. It can also be used for RT level testability analysis to predict architectural testability properties of a netlist. For the validation aspect of the design effort, the VVG-approach overcomes several critical limitations of the current code coverage technique without imposing new penalties. The new RTL code coverage metric can be used for RTL testability analysis and the prediction of gate level fault coverage.

Recent advances in formal verification techniques [16], whose usage is still not widespread, avoid validation vector generation and simulation. These techniques can verify the design without exposing any testability problems. Even when formal verification is employed, VVG-approach will help make the high level design for testability and test generation possible.

Acknowledgment – The authors thank John Vogel and his associates at Cadence Design Systems, Inc. for providing access to *Verifault-XL*, Arthur Friedman for his comments on an early draft, and Natalya Dvorson for developing a C++ program for RTL fault injection.

References

[1] G. Buonanno, F. Ferrandi, and D. Sciuto, "Data-Path Testability Analysis Based on BDDs," in *Proc. IEEE Int'l Symp. Circ. and Syst.*, 1995, pp. 2012–2014.

[2] Cadence Design Systems, Inc., San Jose, CA, *Verifault-XL User Guide*, 1997.

[3] T. Chakraborty and S. Ghosh, "On Behavior Fault Modeling for Combinational Digital Designs," in *Proc. Int'l Test Conference*, 1988, pp. 593–600.

[4] C. H. Chen and P. R. Menon, "An approach to Functional Level Testability Analysis," in *Proc. Int'l Test Conference*, 1989, pp. 373–380.

[5] C. H. Chen and D. G. Saab, "A Novel Behavioral Testability Measure," *IEEE Trans. on Computer Aided Design*, vol. 12, pp. 1960–1970, December 1993.

[6] V. Chickermane, J. Lee, and J. H. Patel, "Design for Testability Using Architectural Descriptions," in *Proc. Int'l Test Conference*, 1992, pp. 752–761.

[7] C. Cho and J. Armstrong, "B-algorithm: A Behavioral Test Generation Algorithm," in *Proc. Int'l Test Conference*, 1994, pp. 968–979.

[8] D. Drako and P. Cohen, "HDL Verification Coverage," *Integrated System Design*, pp. 46–52, June 1998.

[9] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM: Efficient Computation of Observability-Based Metrics for Functional Verification," in *Proc. 35th Annual Design Automation Conf.*, June 1998, pp. 152–157.

[10] I. Ghosh, A. Raghunathan, and N. K. Jha, "A Design for Testability Technique for RTL circuits Using Control/Data Flow Extraction," in *Proc. IEEE/ACM Int'l Conference on Computer-Aided Design*, November 1996, pp. 329–336.

[11] S. Ghosh, "Behavioral-Level Fault Simulation," *IEEE Design and Test of Computers*, vol. 5, pp. 31–42, 1988.

[12] L. H. Goldstein, "Controllability/Observability Analysis of Digital Circuits," *IEEE Trans. Circuits and Systems*, vol. CAS-26, pp. 685–693, September 1979.

[13] P. Jalote, *An Integrated Approach to Software Engineering*. New York: Springer-Verlag, 1991.

[14] M. Kantrowitz and L. M. Noack, "I'm done simulating: Now what? Verification coverage analysis and correctness of checking of the DEC chip 21164 Alpha microprocessors," in *Proc. 33rd Annual Design Automation Conf.*, June 1996, pp. 325–330.

[15] M. Kassab, J. Rajski, and J. Tyszer, "Hierarchical Functional Fault Simulation for High-Level Synthesis," in *Proc. Int'l Test Conference*, 1995, pp. 596–605.

[16] R. P. Kurshan, *Computer-Aided Verification of Coordinating processes*. Princeton, New Jersey: Princeton University Press, 1994.

[17] W. Mao and R. Gulati, "Improving Gate Level Fault Coverage by RTL Fault Grading," in *Proc. Int'l Test Conference*, 1996, pp. 150–159.

[18] P. Sanchez and I. Hidalgo, "System Level Fault Simulation," in *Proc. Int'l Test Conference*, 1996, pp. 732–740.

[19] J. M. Schoen, *Performance and Fault Modeling with VHDL*. Englewood Cliffs, New Jersey: Prentice Hall, 1992.

[20] G. Silberman and I. Spillinger, "Using Functional Fault Simulation and the Difference Fault Model to Estimate Implementation Fault Coverage," *IEEE Transactions on Computer Aided Design*, vol. 9, no. 12, pp. 1335–1343, December 1990.

[21] J. Steensma, W. Geurts, F. Catthoor, and H. Man, "Testability Analysis in High Level Data Path Synthesis," *Journal of Electronic Testing: Theory and Applications*, vol. 4, pp. 43–56, 1993.

[22] J. Stephensen and J. Grason, "A Testability Measure for Register Transfer Level Digital Circuits," in *Proc. Int'l Symp. Fault-Tolerant Computing*, June 1976, pp. 101–107.

[23] E. Sternheim, R. Singh, R. Madhavan, and Y. Trivedi, *Digital Design and Synthesis with Verilog HDL*. San Jose, CA: Automata Publishing Company, 1993.

[24] Synopsys, Inc., Mountainview, CA, *Design Compiler Family Reference Manual*, 1994.

[25] P. A. Thaker, M. E. Zaghloul, and M. B. Amin, "Study of Correlation of Testability Aspects of RTL Description and Resulting Structural Implementations," in *Proc. 12th Int'l Conf. VLSI Design*, January 1999, pp. 256–259.

[26] K. Thearling and J. Abraham, "An Easily Computed Functional Level Testability Measure," in *Proc. Int'l Test Conference*, 1989, pp. 381–390.

[27] TransEDA Ltd., *VeriSure Training Course*, 1997.

[28] T. H. Wang and C. G. Tan, "Practical Code Coverage for Verilog," in *Proc. 4th Int'l Verilog HDL Conf.*, March 1995, pp. 99–104.

Appendix A

An example of the original RTL code follows:

```
module ckt(out1, out2, out3,
           in1, in2, in3, in4);

input[2:0] in1, in2;
input in3, in4;

output[2:0] out1;
output[1:0] out2, out3;

wire[2:0] in1, in2;
reg[2:0] out1;
wire[1:0] out2;
wire[1:0] out3;
wire int_var;

always @(in1 or in2)
  out1[2:0] = in1[2:0] & in2[2:0];

assign int_var = in3 ^ in4;

assign out2[1:0] = (~out1[1:0]) &
                   ({int_var, int_var});

assign out3[1:0] = in3 + int_var;

endmodule
```

The same code modified for fault injection is:

```
module ckt(out1, out2, out3,
           in1, in2, in3, in4);

input[2:0] in1, in2;
input in3, in4;

output[2:0] out1;
output[1:0] out2, out3;

/* Notice change of data types and
   new variables */
wire[2:0] in1, in2;
wire[2:0] in1_a, in2_a;
wire in3, in4;
wire in3_a, in4_a;
wire int_var;
wire int_var_a, int_var_b;
wire[2:0] out1;
reg[2:0] out1_a;
wire[1:0] out1_b;
wire[1:0] out2, out2_a;

always @(in1_a or in2_a)
  out1_a[2:0] = in1_a[2:0] & in2_a[2:0];

assign int_var = in3_a ^ in4_a;

assign out2_a[1:0] = (~out1_b[1:0]) &
                     ({int_var_a, int_var_b});

adder a1(.sum(out3), .sig1(in3),
         .sig2(int_var));

buf in1_a_b0(in1_a[0], in1[0]);
buf in1_a_b1(in1_a[1], in1[1]);
buf in1_a_b2(in1_a[2], in1[2]);

buf in2_a_b0(in2_a[0], in2[0]);
buf in2_a_b1(in2_a[1], in2[1]);
buf in2_a_b2(in2_a[2], in2[2]);

buf in3_a_b0(in3_a, in3);
buf in4_a_b0(in4_a, in4);

buf int_var_a_b0(int_var_a, int_var);
buf int_var_b_b0(int_var_b, int_var);

buf out1_b0(out1[0], out1_a[0]);
buf out1_b1(out1[1], out1_a[1]);
buf out1_b2(out1[2], out1_a[2]);

buf out1_b_b0(out1_b[0], out1_a[0]);
buf out1_b_b1(out1_b[1], out1_a[1]);

buf out2_a_b0(out2[0], out2_a[0]);
buf out2_a_b1(out2[1], out2_a[1]);

endmodule

module adder(sum, sig1, sig2);
input sig1, sig2;
output[1:0] sum;

assign sum[1] = sig1 & sig2;
assign sum[0] = sig1 ^ sig2;

/* Insert RTL faults in this description
same as above*/

endmodule
```

The fault injected RTL code is supplied to *Verifault-XL* for fault simulation to generate VVG. This example shows processing of single and multiple bit input, output and internal variables with single or multiple fan-out. It also shows processing of an arithmetic expression. However, the example shows a very simple scenario. The algorithm is designed to handle much more complex combinations of RTL constructs.

Appendix B

RTL constructs of Verilog HDL [23] supported by VVG fault injection algorithm are described in Table 6.

Table 6: Verilog constructs supported in VVG

RTL Construct Type	RTL Construct
Data Type	reg
	wire
Arrays	one-dimensional array
Basic Operations	+ − *
	> >= < <=
	! && ||
	== !=
	?:
	{ }
	=== !==
	~ & | ^
	<< >>
Procedural Statements	always
	assign
Assignment Operators	blocking (=)
	non-blocking (<=)
Conditional Statements	if-else
	case
	casex
	casez
Event Expression	@
String Substitution	‘define
Parameterization	parameter
Event Control	posedge
	negedge
	level sensitive