Automatic Design Validation Framework for HDL Descriptions via RTL ATPG [*]

Liang Zhang and Michael Hsiao
Department of ECE, Virginia Tech
Blacksburg, VA, 24061, USA
{liang,hsiao}@vt.edu

Indradeep Ghosh
Fujitsu Laboratories of America Inc.
Sunnyvale, CA, 94086, USA
ighosh@fla.fujitsu.com

Abstract

We present a framework for high-level design validation using an efficient register-transfer level (RTL) automatic test pattern generator (ATPG). The RTL ATPG generates the test environments for validation targets, which include variable assignments, conditional statements, and arithmetic expressions in the HDL description. A test environment is a set of conditions that allow for full controllability and observability of the validation target. Each test environment is then translated to validation vectors by filling in the unspecified values in the environment. Since the observability of error effect is naturally handled by our ATPG, our approach is superior to methods that only focus on the excitation of HDL descriptions. The experimental results on ITC99 benchmark circuits and an industrial circuit demonstrate that very high design error coverage can be obtained in a small CPU times.

1. Introduction

Despite of the advances of formal verification methods (such as equivalence checking, model checking, theorem proving, etc.) in recent years, simulation remains the primary approach for design validation, especially at the high abstraction levels of the circuit. The keys to simulation-based validation approach are coverage metrics and vector generation algorithms. The most common coverage metrics are statement, branch and condition coverage adopted from software testing [1]. In [4], the authors proposed a simulation-based ATPG approach for design validation, in which the input VHDL code is first instrumented so that the execution trace can be captured, then the instrumented VHDL is simulated using a commercial simulator. The underlying procedure interacts with the simulator via the trace files and tries to generate vectors to maximize the statement coverage. Another algorithm [9] proposes a mutant analysis and tries to generate vectors to capture every injected mutant. The authors in [10] propose the analysis of paths for input HDL code. Each path starts from a variable definition and ends at usage of that variable. The constraints are formed from the path and provided to constraint solver to generate each validation test. The authors in [3] propose the genetic algorithm (GA) based approach to automatically generate test programs for microprocessor cores. A gate-level fault simulator is used to evaluate the fitnesses of generated test programs and to guide the GA process. The gate-level implementation must be available for this approach. In [6], the authors use integer linear programming and boolean satisfiability methods to generate vectors that achieve high tag (an observability-enhanced statement) coverage [7]. Tag coverage is a better coverage metric than traditional software coverages in that the propagation of error effects to an observation point is considered. In [8], authors proposed a deterministic RTL-ATPG algorithm, which is able to efficiently generate logic-level stuck-at tests from an RTL HDL description. The algorithm utilizes a set of 9-valued algebra to perform symbolic justification and propagation to derive the test environment. Once a test environment is obtained, the precomputed test vectors can be plugged into the test environment to derive the complete set of test vectors. In order to more effectively handle control-intensive circuits, authors in [11] extended the 9-valued algebra to a 10-valued algebra and introduced several high-level heuristics to efficiently use finite state machine (FSM) information to guide the ATPG search process.

While the RTL ATPG algorithm was originally targeted for manufacturing tests, we have discovered that with some modifications, the ATPG algorithm can be extended to generate validation vectors efficiently. The original ATPG targets every construct that is synthesized to the structural RTL netlist. However, for design validation purposes, we only need to target constructs that directly map to variable assignment, arithmetic expressions, and conditional constructs at the behavioral level. The RTL algebra has been augmented to handle embedded counters more efficiently. In addition, in our work, test environment relaxation is used to help validation of additional portions of the design that may be hard to test. As a result, our approach is superior to the test generation algorithm in [8] and is able to generate more compact validation vectors. Our approach is very different from the previous approaches. Most of the previous techniques do not consider

[*]This research was supported in part by NSF grants CCR-0196470, CCR-0305881, and a grant from Fujitsu Labs of America

the observability issues; instead, they focus on maximizing the excitation of the potential error sites. Compared to the approach described in [6], which targets the tag coverage, our test generation method improves upon the tag simulation process and is able to handle much larger designs. Lastly, the back-end test environment translator has been enhanced with to maximize the design error detection. As the result, the generated vectors are more compact and more powerful for design validation.

Note that while automatically deriving validation vectors at the RTL, we are validating on the implementation circuit, which may be a buggy circuit. Nevertheless, this fact can add advantages to our work in the following scenarios:
(i) If a golden RTL model is present, then from the model the test vectors may be derived and used to compare output responses from an implementation.
(ii) The test vectors obtained from an implementation can be applied to its executable behavioral specification (if available) and will produce different outputs when the bugs are excited and propagated in the implementation. (We assume this scenario in this framework)
(iii) The vectors may be used to validate the logic-level circuit derived from an RTL description if formal equivalence checking fails (ie., the RTL description and the gate level implementation are structurally different). Thus, automatically generated validation test benches at the RTL can aid the verification process to a large extent.

The rest of paper is organized as follows: Section 2 presents an overview of our design validation framework. Section 3 describes our validation vector generation algorithm. Section 4 reports the experimental results, and Section 5 concludes the paper.

2. Design Validation Framework

Figure 1 shows the overview of our design validation framework. First, the input HDL description (either VHDL or Verilog) is compiled into an internal structural RTL representation based on Assignment Decision Diagrams (ADD) [2]. Then, the ATPG procedure generates the validation tests and reports the achieved tag coverage. The HDL test bench wrapper is generated to facilitate the simulation. Next, a commercial HDL simulator is invoked to simulate the validation vectors. Finally, the responses are captured and compared against the responses from the specification by the checker program.

2.1. Validation Metrics

In order to measure the thoroughness of the validation, we adopted into our framework the Observability-Based Code Coverage (OBCC or tag coverage)[7]. The OBCC is superior to classical software testing coverages (such as statement and branch coverage) in that it incorporates observability as well as controllability information into the simulation. The basic strategy of OBCC is the efficient computation of tag

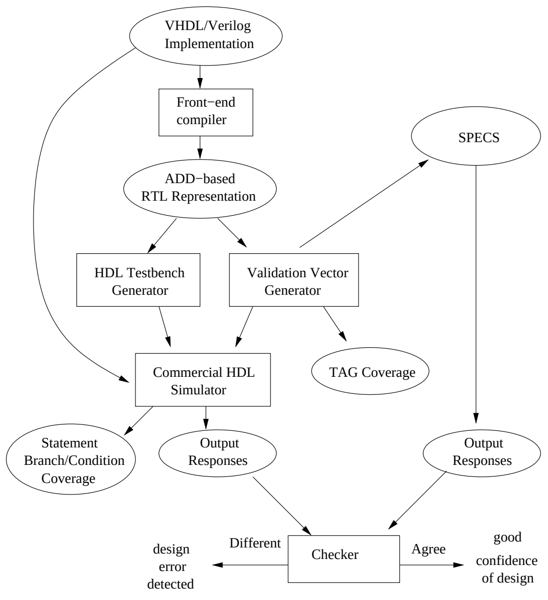

Figure 1. Overview of the design validation flow

Table 1. Δ calculus for an adder

Adder	b	b-Δ	b+Δ	b+U	b+?
a	a+b	a+b-Δ	a+b+Δ	a+b+U	a+b+?
a-Δ	a+b-Δ	a+b-Δ	a+b+?	a+b+?	a+b+?
a+Δ	a+b+Δ	a+b+?	a+b+Δ	a+b+?	a+b+?
a+U	a+b+U	a+b+?	a+b+?	a+b+?	a+b+?
a+?	a+b+?	a+b+?	a+b+?	a+b+?	a+b+?

coverage. The tag is introduced in OBCC as a mechanism to extend standard coverage metrics so that the observability can be computed. A tag is defined as a symbol placed at a given location, which can be used to represent the presence of an incorrect value. First, the tags are injected at variable assignment statements and branch conditions, then the tags are propagated based on a set of calculus rules for supported primitives. We augmented original tag calculus with unsigned (U) tag. Table 1 shows the augmented tag calculus for an adder. The +Δ represents a positive tag, while -Δ denotes a negative one. The ? means that tag may be killed as the result of operation during execution. A tag is declared to be observed only when either +Δ ,-Δ, or U has been successfully propagated to at least one PO.

3. Validation Vector Generation Algorithm

The core of our validation framework is an efficient RTL ATPG tool, which generates the validation vectors for a given HDL description. The justification and propagation of controllability and observability objectives are carried out symbolically using a set of 10-valued RTL algebra. The 10-valued RTL algebra, first proposed in [8], then extended in [11], includes following symbols:

- **Cg** (general controllability) is the ability to control a variable to arbitrary value.
- **C0** (controllability to zero) is the ability to control a variable to the value 0.
- **C1** (controllability to one) is the ability to control a variable to the value 1; i.e., ”000...01”.
- **Ca1** (controllability to all ones) is the ability to control the variable to all ones; i.e., ”111...11”.
- **Cq** (controllability to a constant) is the ability to control the variable to any fixed constant.
- **Cz** (controllability to the Z value) is the ability to control the variable to high-impedance Z.
- **Cs** (controllability to a state) is the ability to control the state variable to a particular state.
- **Cp[a,b]** (controllability to a particular range) is the ability to control the variable within the range of [a,b].
- **O** (observability) is the ability to observe a fault at a variable.
- **O’** (complement observability) is defined for single-bit variables only. It signifies the zero/one fault.

Figure 2 shows the test generation flow. First, a preprocessor builds a validation target list for the circuit, which includes all condition, arithmetic, and assignment constructs. Next, the ATPG iterates through the list and generates the test environment for each target. If the test environment cannot be obtained for a given validation target, the ATPG tries to generate the relaxed test environment for it. After the test environment generation stops, the back-end translator is invoked to generate the validation vectors from the test environments.

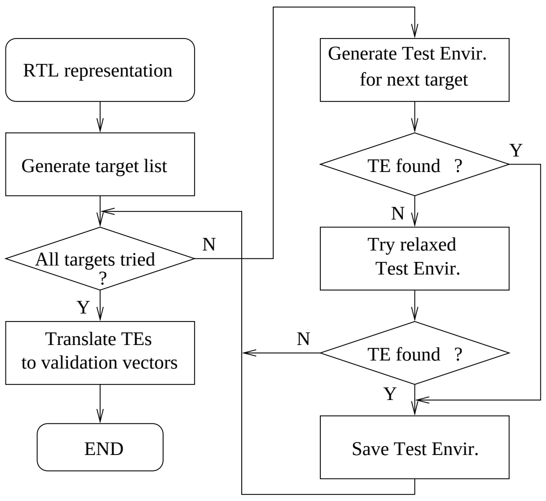

Figure 2. Test Generation Flow

3.1. Test Environment Generation

The test environment is a set of conditions that allow controllability and observability of the validation target. Each test environment can be viewed as a symbolic path which starts from the PIs, traversing through the target site, and reaches at one or more POs or observable variables. The test environment generation process, as shown in Figure 3, is essentially searching for a sufficient symbolic path, through which the excitation objectives can be delivered to the target site, and error effect can be propagated to the PO.

```
generate_test_environment() {
   while(select_symbolic_path() == TRUE) {
      inject_symbolic_propagation_objectives();
      inject_symbolic_excitation_objectives();
      if (justify_all_objectives()==TRUE) {
        save_test_environment();
         exit();
      }
   }
}
```

Figure 3. ATPG Algorithm

Consider the VHDL description and its structural RTL in Figure 4 as an example to illustrate the above algorithm. Assume that *RST*, *ina* and *inb* are PIs, and *out* is the only PO

```
If RST = 1 then
      a   := 1;
      b   := 1;
      c   := 1;
      i    := 1;
      out := 1;
elsif clk'event and clk = 1 then
      i := i + 1;
      if ( i = 2) then
            a := ina - 7;
            b := inb;
      elsif (i = 3) then
            c := a * b;
      elsif (i = 4) then
            out := c;
      endif;
endif;
```

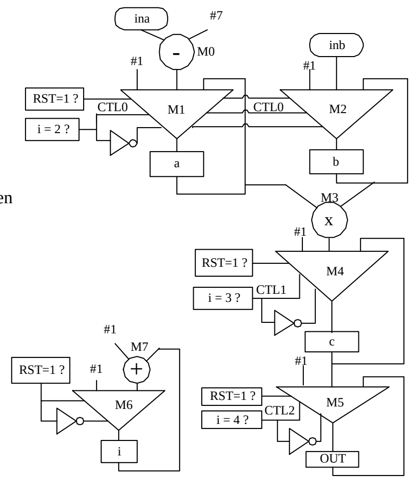

Figure 4. Sample VHDL Code and Structural RTL

for the circuit. Suppose that the multiplier M3 is our current validation target. Our algorithm executes as follows:

1. Find the shortest propagation path $M3 \rightarrow M4 \rightarrow C \rightarrow M5 \rightarrow OUT$.
2. Propagation constraints of (CTL1,1,C1) and (CTL2,2,C1) are injected. The first objective means a C1 algebra at time frame 1 is needed on signal CTL1.

3. Excitation objectives of (a,0,Cg) and (b,0,Cg) are injected at the inputs of M1. The objectives mean algebra Cg is needed on both signal a and b at time frame 0.
4. Justify all objectives via a branch-and-bound search.
5. All objectives are justified. The test environment is generated as shown below.

```
(RST,-2,C1);
(RST,-1,C0);
(ina,-1,Cg);
(M0 ,-1,Cg):  (ina,-1,Cg);
(M1 , 0,Cg):  (M0 ,-1,Cg);
(a  , 0,Cg):  (M1 , 0,Cg);
(inb,-1,Cg);
(M2 , 0,Cg):  (inb,-1,Cg);
(b  , 0,Cg):  (M2 , 0,Cg);
(RST, 0,C0);
(RST, 1,C0);
(RST, 2,C0);
```

Note that the test environment contains all justified algebra on the PIs and Cgs on internal nodes. The fan-in nodes are also included in test environment for the internal nodes.

3.2. Test Environment Translation

A generated test environment must be translated into validation vector(s) to be applicable to the design. Table 2 highlights partial results as the procedure proceeds. Suppose we need to apply 11 and 4 at the two inputs of the multiplier. First, all algebra except the C_g at PIs are translated. For the above test environment, only the value of *RST* is determined at this step, as shown from the columns under the heading "STEP I". Secondly, the value 11 is plugged into (a,0), then following the trace $(a, 0, Cg) \rightarrow (M1, 0, Cg) \rightarrow (M0, -1, Cg) \rightarrow (M0, -1, Cg) \rightarrow (ina, -1, Cg)$ the value can be propagation backward to the PI. The value may need to be adjusted when propagated through certain types of RTL constructs. For example, the value 11 is propagated from (a,0) through (M0,-1) without any adjustment. However the 11 on (M0,-1) implies the 18 at (*ina*,-1), since M0 is a subtracter, and the other operand is constant 7. Similarly the 4 can be plugged in (b,0,Cg) and value can be propagated to the (*inb*,-1). The translation results are recorded in columns under "STEP II". Finally, all unspecified PIs are filled with the random numbers to form the fully specified test vectors. The last three columns show the final test vectors.

3.3. Test Environment Relaxation

If the complete test environment cannot be derived using the above algorithm, the ATPG relaxes the controllability condition and repeats the algorithm to generate a relaxed test environment. The benefit of test environment relaxation can be illustrated by following example.

Table 2. Test Environment Translation

Time	STEP I			STEP II			STEP III		
Frame	RST	ina	inb	RST	ina	inb	RST	ina	inb
-2	1	x	x	1	x	x	1	0	5
-1	0	x	x	0	18	4	0	18	4
0	0	x	x	0	x	x	0	1	0
1	0	x	x	0	x	x	0	3	8
2	0	x	x	0	x	x	0	9	3

1. $c := a + b;$
2. $c := a + d;$
3. $c := a - b;$

Suppose statement 1 is the correct implementation, while statements 2 and 3 are erroneous versions. With complete test environment for statement 1, we can fully control the values of a and b. In other words, by enforcing the values of a and b to be different from any other signals, the erroneous values computed for c in statements 2 and 3 can be guaranteed to be different from correct one. The errors will be propagated to PO by the test environment and the design error will be captured.

Now suppose that the ATPG cannot find a complete test environment for statement 1, it produces a relaxed version instead. This relaxed test environment does not guarantee full controllability over the operands, the detection of statement 3 can still be assured as long as the value of b is not zero. However, the relaxed test environment should not replace the complete test environment always, since it can only conditionally detect error in statement 2. The detection is contingent upon values for b and d be different.

3.4. Techniques to Maximize the Error Detection

In the previous example, if back-end translator accidentally sets the value of b equal to the value of d, the generated validation vectors will not detect the erroneous implementation of statement 2. To remedy this problem, while translating the test environment, the ATPG keeps a list of values that have been assigned to the signals and enforces exclusiveness of signal values as much as possible so that the signal substitution errors will be detected.

For the relational operations, we maximize the error detection by plugging 3 properly selected values into test environment. The signal values can be determined as follows:

1. If the value of one input signal is fixed to k, then apply 3 values of $k + 1$, k, and $k - 1$ to the other input signal.
2. If no input is fixed, first select a unique value to one input, then follow step 1 for the other input.

For example, in validating the condition $(a > 3)$, we need to check for (1) the correct use of "greater-than" operator here, and (2) value 3 is the correct boundary of the condition. In other words, we need to differentiate it from the following implementations, where k stands for any value, and n stands for any value but 3:

$$(a \geq k),\ (a \neq k),\ (a < k),\ (a \leq k),\ \text{and}\ (a > n)$$

Following the above mentioned rules, we know that 4, 3, and 2 need to be plugged into the generated test environment for signal a to obtain 3 different validation sequences. Although none of the 3 sequences can individually detect all the bugs, collectively they can capture all bugs. Note that, from the tag coverage [6] point of view, requiring all three validation sequences on $(a > 3)$ is an overkill because for each condition (or statement), two sequences are sufficient to capture both positive and negative tags on it. As the result, the [6] cannot guarantee the error detection associated with operational operators.

4. Experiments

We applied our framework to 10 ITC99 [5] benchmark circuits, as well as *GPIO*, an industrial general purpose input output bus controller, on a 2.0 GHz Pentium-4, with 512 MB RAM, running the Linux operating system. For each VHDL description, we manually injected 15 to 30 bugs, which include the most typical design errors, such as missing case statement, missing signals, wrong signals, wrong variable values, wrong ordering of nested if statements, wrong operation types, and etc.

Table 3 shows the circuit characteristics and our RTL ATPG results. Note that, our framework works on a given HDL description, and no gate-level implementations are needed. However, we include the gate-level characteristics in the table to show the complexities of each design. For each circuit, the total number of VHDL lines is first reported, followed by the number of logic gates corresponding to the VHDL and the number of flip-flops. Then, the number of validation vectors generated using our RTL ATPG and the execution time (on a Pentium 4 2.0 GHz Linux machine) are reported. For example, circuit B11 has 118 lines of VHDL code and can be synthesized to 397 gates at the gate level, containing 30 FFs. Using the VHDL alone for our RTL ATPG, 193 validation vectors were generated in only 1.46 seconds.

Note that the test generation times of our method are orders of magnitude smaller than the ones reported in [6] for similar sized circuits. A direct comparison is not possible as the circuits used in [6] are not publicly available.

Figures 5 and 6 show the tag coverage and bug coverage, respectively, for each circuit. Both random validation and our approach are reported, and the bug coverage is defined as the ratio of the number of detected bugs to the total number of injected bugs. For the random approach, 5000 random vectors were applied to each circuit.

For all circuits, our RTL ATPG outperformed random generation of validation vectors. In fact, in the five largest circuits, our method achieved orders of magnitude better results in both tag and bug coverages. Note that since the number of validation vectors we generated was significantly fewer than 5000, the validation time would be reduced by the same ratio. For example, in circuit B08, with only 100 vectors, we were able to achieve 93.3% tag coverage in under 1 seconds of computation, while with 5000 random vectors, only 2.2% tag coverage was achieved. In terms of simulation time for validation, 50-fold improvement was achieved. In fact, increasing the random vectors beyond 5000 vectors still would not be helpful. In this regard, several orders of magnitude lower validation time can be achieved with our method.

Table 3. Experiment results of Our Approach

Circuit	Characteristics			Our Approach	
	#lines	#Gates	#FFs	#Vectors	TGen(s)
B01	110	56	5	192	0.02
B02	70	30	4	77	0.01
B03	141	181	30	653	0.07
B04	102	547	66	165	0.02
B05	332	643	34	255	1.89
B06	128	76	9	49	0.02
B07	92	434	51	104	0.04
B08	89	190	21	100	0.28
B10	167	190	17	337	0.8
B11	118	397	30	193	1.46
GPIO	1002	1720	148	652	2.31

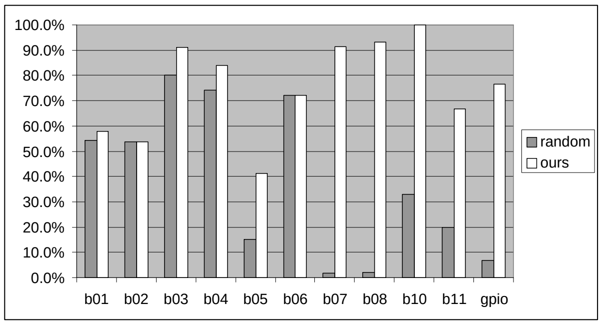

Figure 5. Tag Coverages

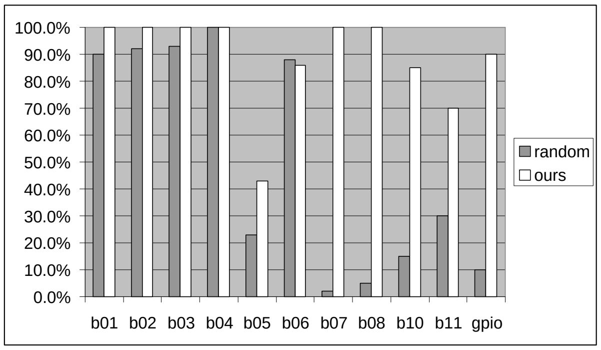

Figure 6. Bug Coverages

Figure 7 shows the detailed validation results on the industrial circuit *GPIO*. The X-axis is the number of validation vectors applied, while the Y-axis is the coverage of respective metrics. 30 design errors were injected in this circuit. Note that *GPIO* is not randomly validatable, as shown by the curves for random vectors. Our approach was able to obtain both high tag coverage as well as high bug coverage. We can also clearly observe the close correlation between the achieved tag coverage and bug coverage for both random vectors and vectors generated by our approach. However, the tag coverage is slightly more pessimistic than the bug coverage in that the coverage reported is lower. In other words, the actual bug coverage generally is higher than the tag coverage. Nevertheless, these two coverage metrics track very well with each other.

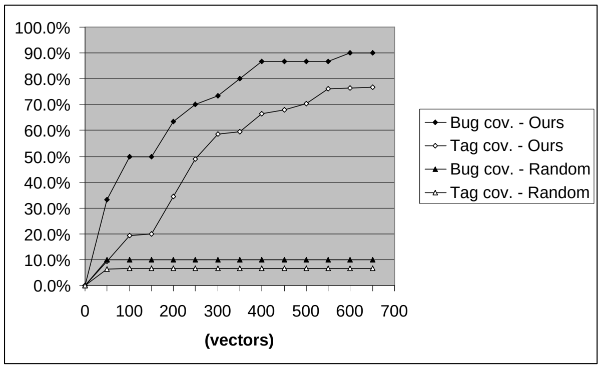

Figure 7. Coverages on GPIO

We also ran the vectors generated by our approach with TRansEDA[12] coverage analysis tool to obtain the statement, branch, conditional and tag coverages. The results are reported in Table 4. We can see that for most circuits, the three traditional coverages (statement, branch, condition) are overly optimistic measures, and that tag coverage is superior in that it reflects the bug coverage better. In B05, due to large redundancies in both VHDL and gate-level implementations, the coverages are low. However, in most other circuits, such as GPIO, the 100% measures would provide little confidence on the effectiveness of validation vectors. Note that these traditional coverage metrics are currently used in the industry to measure the level of design validation and a number of commercial tools exist in this effect.

Table 4. Coverages of Our Approach

Circuit	State. (%)	Branch (%)	Cond.(%)	Tag(%)
B01	100	100	98.4	58.3
B02	100	100	100	53.8
B03	100	100	100	90.9
B04	100	100	100	83.8
B05	65.4	61	n/a	41.1
B06	100	100	100	72.1
B07	97.9	97.2	n/a	91.2
B08	100	100	n/a	93.3
B10	94.7	95.3	90	100
B11	85.7	87.9	100	66.7
GPIO	100	100	100	76.6

5. Conclusion

We have presented an automatic design validation framework for HDL descriptions. The core of our framework is a modified RTL ATPG algorithm, which efficiently generates the validation vectors. Our approach is superior to existing approaches that target only the excitation while ignoring propagation of the errors in the design. We also allow for relaxation of test environments such that additional hard errors may be detected. Experiments show that our approach is able to generate high quality validation vectors, which achieve both high tag coverage and high bug coverage with very low computational cost. Orders of magnitude improvement in coverage over random patterns were achieved.

References

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

[2] V. Chaiyakul, D. D. Gajski, and L. Ramachandran. ”High-level Transformations for Minimizing Syntactic Variances”, In *Proc. Design Automation Conf.*, pages 413–418, June 1993.

[3] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero. ”Fully Automatic Test Program Generation for Microprocessor Cores”, In *Pro. Design, Automation and Test in Europe*, pages 1006–1011, 2003.

[4] F. Corno, M. Reorda, G. Squillero, A. Manzone, and A. Pincetti. ”Automatic Test Bench Generation for Validation of RT-level descriptions: an industrial experience”, In *Proc. DATE*, pages 385–389, 2000.

[5] F. Corno, M. S. Reorda, and G. Squillero. ”RT-level ITC’99 benchmarks and first ATPG results”, *IEEE Design & Test of Computers*, 17(3):44–53, July-September 2000.

[6] F. Fallah, P. Ashar, and S. Devadas. ”Simulation vector generation from HDL descriptions for observability-enhanced statement coverage”, In *Proc. Design Automation Conference*, pages 666–671, 1999.

[7] F. Fallah, S. Devadas, and K. Keutzer. ”OCCOM – efficient computation of observability-based code coverage metrics for functional verification”, *IEEE Trans. Computer-Aided Design*, 20(8):1003–1015, August 2001.

[8] I. Ghosh and M. Fujita. ”Automatic Test Pattern Generation for Functional Register-Transfer level Circuits Using Assignment Decision Diagrams”, *IEEE Trans. Computer-Aided Design*, 20(3):402–415, March 2001.

[9] G. Hayek and C. Robach. ”From specification validation to hardware testing: A unified method”, In *Proc. International Test Conference*, pages 885–893, 1996.

[10] C. Paoli, M. Nivet, and J. Santucci. ”Use of constraint solving in order to generate test vectors for behavioral validation”, In *Proc. High Level Design Validation and Test Workshop*, pages 15–20, 2000.

[11] L. Zhang, I. Ghosh, and M. Hsiao. ”Efficient sequential ATPG for functional RTL circuits”, To appear in *Proc. International Test conference*, 2003.

[12] VN-COVER, *http://www.transeda.com*