Functional Vector Generation for Sequential HDL Models Under an Observability-Based Code Coverage Metric

Farzan Fallah, Pranav Ashar, and Srinivas Devadas

***Abstract*—Design validation and verification is the process of ensuring correctness of a design described at different levels of abstraction during the design process. Design validation is the main bottleneck in improving design turnaround time. Currently, simulation is the primary methodology for validation of the first description of a design. In this paper we integrate directed search methods and observability-based code coverage metric (OCCOM) computation into an algorithm for generating test vectors under OCCOM for sequential HDL models. A prototype system for design validation under OCCOM has been built. The system uses repeated coverage computation to minimize the number of vectors generated. Experimental results using the test vector generation system are presented.**

***Index Terms*—Coverage metric, HSAT, observability, observability-based code coverage metric (OCCOM), satisfiability, test vector, validation.**

I. Introduction

Validation is by far the most time-consuming task in the design of microchips, and validation time influences the time-to-market. In order to validate an HDL description, simulation is usually used.

In order to do simulation, it is necessary to have some test vectors. Because trying all possible test vectors is not practical, only a small number of test vectors have to be chosen. This raises the question of how good the test vectors that we are using are and how well the design has been tested. In order to answer these questions, a coverage metric can be used, and the design can be simulated under some test vectors to achieve the targeted coverage under the chosen metric. Some coverage metrics have been used so far are transition coverage and coverage of all statements, branches, and paths borrowed from software testing [1].

Covering all statements in an HDL code [2] or covering all branches are tractable, but they are not sufficient for validating a design. Coverage of all paths in an HDL code or the coverage of all transitions in an FSM model of an implementation of a design [3] result in many test vectors and are not practical.

Most of these metrics are based on activation of statements and do not guarantee detection of a possible error in the statement. To detect an error, an erroneous value for one of the outputs of the design must be observed. We use an observability-based code coverage metric (OCCOM) [4] in our system to address the observability requirement.

After choosing a coverage metric, test vectors should be generated. Manual test vector generation can result in very good test vectors but it is time consuming. Random methods can generate test vectors quickly but the quality of the test vectors may not be satisfactory. As a result, an automatic deterministic test vector generator is desired.

In this paper, we integrate the hybrid satisfiability test vector generation algorithm (HSAT) introduced in [5] with OCCOM. We perform coverage analysis after generating each test vector to find out the portions of the design covered by the test vector. This enables us to generate test vectors for portions uncovered by the previous test vectors only. Hence, minimizing the total number of generated test vectors. Our system is better suited to target portions of a design that cannot be covered by random and heuristic-based test vector generation methods. Due to using a complete method for generating test vectors, our method cannot handle very large and deep sequential designs.

The reader is referred to [4] and [5] for details of the observability-based code coverage metric and the hybrid satisfiability test generation algorithm, respectively.

Details of our vector generation algorithm are presented in Section II, while our test vector generation algorithm for sequential circuits is described in Section III. Section IV presents the experimental results of using our test vector generator.

Manuscript received January 7, 2002; revised April 22, 2002.

F. Fallah is with the Fujitsu Laboratories of America, Inc., Sunnyvale, CA 94085 USA.

P. Ashar is with the CCRL NEC USA, Princeton, NJ 08540 USA.

S. Devadas is with the Laboratory for Computer Science, Massachusetts Institute of Technology (MIT), Cambridge, MA 02139 USA.

Digital Object Identifier 10.1109/TVLSI.2002.808438

II. Vector Generation Algorithm for OCCOM

Details of a vector generation algorithm for OCCOM are given in this section. The goal of the algorithm is to generate a vector so that a tag injected in a specific line of an HDL model is propagated to one of the outputs of the model.

A. The Basic Algorithm

The basic algorithm operates according to the following steps.

1) An assignment in the HDL model is selected to inject a tag. The variable in the left-hand side of the selected assignment is represented by V.
2) HSAT constraints are written for the HDL model. The set of constraints are called error-free constraints.
3) HSAT constraints are written for the tag-injected HDL model. Every variable Y is replaced by $\mathrm{tagged\_}Y$. This set of constraints are called tagged constraints.
4) For every input of the HDL model the following constraint is added to the set of constraints,
$$\mathrm{tagged\_}input = input.$$
5) Tag Injection: In the constraints corresponding to the selected assignment for tag injection, $\mathrm{tagged\_}V$ is replaced by $\mathrm{tagged\_}V - \Delta$, or $\mathrm{tagged\_}V + \Delta$, depending on the sign of the tag.
6) Constraints for detecting the tag on at least one of the output variables, namely observability constraints, are added. For example, if variables $O1$ and $O2$ are the outputs of the HDL model, the following constraints will be added,
$$(\mathrm{tagged\_}O1 > O1) \vee (\mathrm{tagged\_}O1 < O1) \vee$$
$$(\mathrm{tagged\_}O2 > O2) \vee (\mathrm{tagged\_}O2 < O2).$$
7) The lower and the upper bounds for the variables are written.
8) The set of HSAT constraints are solved using the technique described in **[5]**.

If there is a solution to the HSAT problem, the resulting values for the inputs can be used as a test vector, and the generated test vector can propagate the tag injected in the selected line to one of the outputs. If there is no solution to the HSAT problem, there is no test vector with the desired property.

B. Example Application of the Basic Algorithm

Consider the following Verilog code as an example to illustrate the previous algorithm:

```
module test(clk, a, b, in1, in2, out)
input clk;
input [7:0] a, b;
```

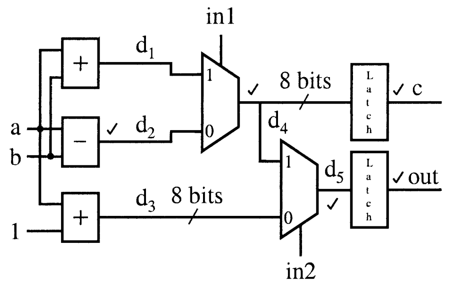

Fig. 1. Structural RTL and marked variables for Section II-B example.

```
input in1, in2;
output [7:0] out;
reg [7:0] c, out;
always(@posedge clk)
 begin
  if(in1)
   c = a + b;
  else c = a - b;
  if(in2)
   out = c;
  else out = a + 1;
 end
endmodule
```

If $in1 = 1$ and $in2 = 1$, *out* will be equal to $a + b$. Having $in1 = 0$ and $in2 = 1$ results in the value $a - b$ for out. Finally, if $in2 = 0$, out will be equal to $a + 1$. The circuit generated from this code is shown in Fig. 1.

The objective is to detect a tag injected in the $else$ clause of the first if statement in the output. The following constraints are generated for the various statements (indicated in comments below) in the error-free version of the circuit.

`// if(in1) c = a + b`
$d1 - a - b = 0$
$c - d1 + 256(1 - in1) \geq 0$
$c - d1 - 256(1 - in1) \leq 0$
`// if/else c = a - b`
$d2 - a + b = 0$
$c - d2 + 256\ \ in1 \geq 0$
$c - d2 - 256\ \ in1 \leq 0$
`// if(in2) out = c`
$out - c + 256(1 - in2) \geq 0$
$out - c - 256(1 - in2) \leq 0$
`// if/else out = a + 1`
$d3 - a = 1$
$out - d3 + 256\ \ in2 \geq 0$
$out - d3 - 256\ \ in2 \leq 0$

To see how the above constraints model the circuit, consider the first set of constraints corresponding to `if(in1) c = a + b`. The equality $d1 - a - b = 0$ models the adder in the `then` clause of the `if` statement. The behavior of the multiplexor corresponding to the `then` clause when the control signal of the multiplexor is 1 (i.e., $in1 = 1$) is modeled using two inequality constraints. Assuming the value of $in1$ is 1, the constraints are simplified to,

`// if(in1) c = a + b`
$c - d1 \geq 0$
$c - d1 \leq 0$

This implies $c = d1$. If the value of $in1$ is 0, the constraints are simplified to,

`// if(in1) c = a + b`
$c - d1 \geq -256$
$c - d1 \leq 256$

These two constraints are always satisfied because of the assumption on the number of bits of variables. As a result, the value of variable c is defined by the constraints corresponding to `if/else c = a - b`.

In the next step, constraints are written for the circuit with the error,

`// if(in1) c = a + b`
$\text{tagged\_}d1 - \text{tagged\_}a - \text{tagged\_}b = 0$
$\text{tagged\_}c - \text{tagged\_}d1 + 256(1 - \text{tagged\_}in1) \geq 0$
$\text{tagged\_}c - \text{tagged\_}d1 - 256(1 - \text{tagged\_}in1) \leq 0$
`// if/else c = a - b`
$\text{tagged\_}d2 - \text{tagged\_}a + \text{tagged\_}b = 0$
$\text{tagged\_}c - \text{tagged\_}d2 + 256\ \ \text{tagged\_}in1 \geq 0$
$\text{tagged\_}c - \text{tagged\_}d2 - 256\ \ \text{tagged\_}in1 \leq 0$
`// if(in2) out = c`
$\text{tagged\_}out - \text{tagged\_}c + 256(1 - \text{tagged\_}in2) \geq 0$
$\text{tagged\_}out - \text{tagged\_}c - 256(1 - \text{tagged\_}in2) \leq 0$
`// if/else out = a + 1`
$\text{tagged\_}d3 - \text{tagged\_}a = 1$
$\text{tagged\_}out - \text{tagged\_}d3 + 256\ \ \text{tagged\_}in2 \geq 0$
$\text{tagged\_}out - \text{tagged\_}d3 - 256\ \ \text{tagged\_}in2 \leq 0$

The values of the inputs for the error-free and tagged models have to be equal. Consequently, the following constraints have to be added to the set of constraints:

$$\begin{aligned} a &= \text{tagged\_}a \quad & in1 &= \text{tagged\_}in1 \\ b &= \text{tagged\_}b \quad & in2 &= \text{tagged\_}in2. \end{aligned}$$

In the next step, a tag is injected in the selected line, and the corresponding constraint in the erroneous circuit is replaced by,

$$\text{tagged\_}d2 - \Delta - \text{tagged\_}a + \text{tagged\_}b = 0.$$

To guarantee the detection of the tag in the output, the following constraints are added,

$$(\text{tagged\_}out > out) \vee (\text{tagged\_}out < out).$$

The lower and the upper bounds for the variables are,

$$\begin{array}{ll} 0 \leq a \ \leq 255 & 0 \leq \text{tagged\_}b \leq 255 \\ 0 \leq b \ \leq 255 & 0 \leq \text{tagged\_}c \leq 255 \\ 0 \leq c \ \leq 255 & 0 \leq \text{tagged\_}out \leq 255 \\ 0 \leq \ out \leq 255 & 0 \leq \text{tagged\_}in1 \leq 1 \\ 0 \leq \ in1 \leq 1 & 0 \leq \text{tagged\_}in2 \leq 1 \\ 0 \leq \ in2 \leq 1 & 0 \leq \text{tagged\_}d1 \leq 255 \\ 0 \leq \ d1 \leq 255 & 0 \leq \text{tagged\_}d2 \leq 255 \\ 0 \leq \ d2 \leq 255 & 0 \leq \text{tagged\_}d3 \leq 255 \\ 0 \leq \ d3 \leq 255 & -255 \leq \Delta \leq 255 \\ 0 \leq \ \text{tagged\_}a \leq 255. & \end{array}$$

After solving the constraints, the resulting values can be used as a test vector to activate and propagate the tag injected in the $else$ branch of the first if statement to the output.

C. Decreasing the Size of the HSAT Problem

Duplicating every constraint and using a new variable for every variable in the error-free model results in an HSAT problem twice as big as the original one. In many cases it is possible to decrease the size of the problem easily. Consider the circuit of Fig. 1. In the figure the tag injection has been shown by marking the output of the subtractor.

Injecting a tag means that the values of the variables in the left-hand-side of the selected HDL line in the erroneous model and the error-free model are different. The effect of that difference can be propagated only to the variables in the transitive fanout of the tagged variable. The transitive fanout is defined recursively as follows, for a primary output it is the null set. For other variables it is defined as the union of the fanout of the variable and the transitive fanout of every variable in the fanout of the original variable. Variables in the transitive fanout of the output of the subtractor have been marked in Fig. 1.

In order to write the equations for the erroneous model, it is necessary to duplicate only the marked variables and the constraints which have at least one marked variable.

Using these facts, the new HSAT problem will be:

`// if(in1) c = a + b`
$d1 - a - b = 0$
$c - d1 + 256(1 - in1) \geq 0$
$c - d1 - 256(1 - in1) \leq 0$
`// if/else c = a - b`
$d2 - a + b = 0$
$c - d2 + 256\ \ in1 \geq 0$
$c - d2 - 256\ \ in1 \leq 0$
`// if(in2) out = c`
$out - c + 256(1 - in2) \geq 0$
$out - c - 256(1 - in2) \leq 0$
`// if/else out = a + 1`
$d3 - a - 1 = 0$
$out - d3 + 256\ \ in2 \geq 0$
$out - d3 - 256\ \ in2 \leq 0$
`// if(in1) c = a + b`
$\text{tagged\_}c - d1 + 256(1 - in1) \geq 0$
$\text{tagged\_}c - d1 - 256(1 - in1) \leq 0$
`// if/else c = a - b`
$\text{tagged\_}d2 - \Delta - a + b = 0$
$\text{tagged\_}c - \text{tagged\_}d2 + 256\ \ in1 \geq 0$
$\text{tagged\_}c - \text{tagged\_}d2 - 256\ \ in1 \leq 0$
`// if(in2) out = c`
$\text{tagged\_}out - \text{tagged\_}c + 256(1 - in2) \geq 0$
$\text{tagged\_}out - \text{tagged\_}c - 256(1 - in2) \leq 0$
`// if/else out = a + 1`
$\text{tagged\_}out - d3 + 256\ \ in2 \geq 0$
$\text{tagged\_}out - d3 - 256\ \ in2 \leq 0$
`// Tag detection`
$(\text{tagged\_}out > out) \vee (\text{tagged\_}out < out)$
`// Variables bounds`
$0 \leq a \leq 255$
$0 \leq b \leq 255$
$0 \leq c \leq 255$
$0 \leq out \leq 255$
$0 \leq in1 \leq 1$
$0 \leq in2 \leq 1$
$0 \leq d1 \leq 255$
$0 \leq d2 \leq 255$
$0 \leq d3 \leq 255$
$0 \leq \text{tagged\_}c \leq 255$
$0 \leq \text{tagged\_}out \leq 255$
$0 \leq \text{tagged\_}d2 \leq 255$
$-255 \leq \Delta \leq 255$

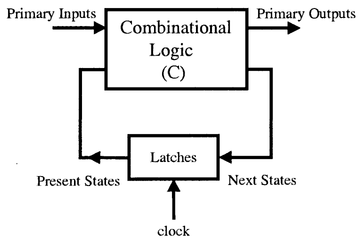

Fig. 2. General synchronous sequential circuit.

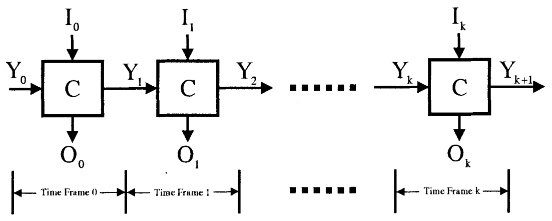

Fig. 3. The iterative logic array of a sequential circuit.

As one can see, the number of variables has decreased from 19 to 13, and the number of constraints has decreased from 28 to 21.

D. Making Test Vector Generation Consistent With the OCCOM Analysis

Although the test vector generated using the described method can propagate the injected tag to one of the outputs of the circuit, the OCCOM analysis explained in [4] may report the tag undetectable. The reason is that the OCCOM analysis is based on the likelihood of the propagation of a tag; it does not use the tag magnitude information and it considers tag cancellation. On the other hand, the test vector generation algorithm generates a test vector which propagates a tag with a specific magnitude to one of the outputs. To make the test vector generation and coverage analysis consistent, it is possible to make the OCCOM analysis more accurate by using the tag's magnitude information. As another solution, it is possible to select a path which can propagate the tag according to the OCCOM tag calculus and use the algorithm described in [5] to generate a test vector for propagating the tag through the selected path.

III. Test Vector Generation for Sequential HDL Models

A. Time-Frame Expansion Method

We use the time-frame expansion method to generate test vectors for sequential HDL models. Fig. 2 shows a sequential circuit.

A synchronous sequential circuit can be modeled by a combinational circuit, called the iterative logic array. The process of modeling the sequential circuit using an iterative logic array is called time-frame expansion.

Fig. 3 shows the iterative logic array of a sequential circuit. Assuming the circuit is in the initial state Y_0 and the input sequence $\{I_0, I_1, \ldots, I_k\}$ is applied to the circuit, the next state sequence will be $\{Y_1, Y_2, \ldots, Y_{k+1}\}$ and the output sequence will be $\{O_0, O_1, \ldots, O_k\}$.

After modeling a sequential circuit with a combinational one, a test vector is generated for the combinational circuit. The basic algorithm operates according to the following steps.

1) An assignment in the HDL model is selected to inject the tag. The tag is injected in all time frames. The variable in the left-hand-side of the selected assignment is represented by V.

2) An upper bound on the number of time frames, $t_{\max}$, that will be used for vector generation is selected.[1]
3) t denotes the number of time frames that the design will be expanded to in the current attempt. t is set to one.
4) The HDL model is unrolled t times.
5) HSAT constraints for both error-free and erroneous versions of the HDL model are generated according to the techniques described in **Section II**.
6) The HSAT problem is solved using the algorithm described in **[5]**.
7) If there is no solution to the HSAT problem and $t < t_{\max}$, t is incremented by 1 and the algorithm reverts to Step 4.
8) If there is no solution to the HSAT problem and $t = t_{\max}$, the algorithm returns reporting the tag cannot be covered within $t_{\max}$ time frames.
9) If there is a solution to the HSAT problem, it is reported as the desired test vector.
10) The test vector for the iterative logic array is converted to a test sequence for the sequential HDL model.

B. Clause Ordering

It is possible to use some structural properties of a design to order Boolean clauses in a way that the HSAT solver will solve the HSAT problem more efficiently. A good clause ordering will help the HSAT solver in finding a good candidate variable during the branch-and-bound process.

Depth-first ordering tries to keep the clauses corresponding to a path from a tag to the outputs/latches together. This way, the HSAT solver will branch on variables corresponding to a path in the circuit.

In topological ordering, clauses corresponding to an operator appear after the clauses corresponding to all the operators in its transitive fanin. The transitive fanin of an operator op is defined as the union of all operators whose transitive fanout contain op. This way the HSAT solver is likely to set the values of the inputs of an operator early in the search. In turn, this is likely to imply a value on the output of the operator.

C. Improving the Performance of the Algorithm

Unrolling HDL models increases the size of HSAT problems rapidly. This can degrade the speed of test vector generation substantially. It can easily be seen that in order to detect a tag in the output, the tag should be propagated through a path consisting of marked variables. In order to simplify the search for a solution, information about paths can be added to an HSAT problem.

As an example, in Fig. 1, in order to have a tag on out, it is necessary to have $in1 = 1$ and $in2 = 1$. Adding this constraint to the original HSAT problem can help the HSAT solver to find a solution to the problem more quickly. In the previous example, there was only one path between the injected tag and out, but in general there are many paths. As a result propagation constraints will be in disjunctive normal form. Transforming propagation constraints to conjunctive normal form which is appropriate for the HSAT solver can result in an exponential growth in the number of the clauses or will require the addition of several intermediate variables. As a result, it is not always practical to use them.

In order to make this approach practical, it is possible to generate propagation constraints only for a limited number of paths between the injection point and each variable. This controls the size of the HSAT problem that must be solved at any given time, usually leading to a shorter run time. If the HSAT problem is found to be infeasible, another set of paths is chosen. As a variation on this, short paths are selected when the HSAT problem is very large. On the other hand, preference is given to long paths when it is important to cover as many tags as possible with each vector to minimize the number of generated vectors.

[1]The only known theoretical bound on the required number of time frames is exponential in terms of the number of latches of the circuit and is not tight enough for practical purposes.

D. Maximizing the Tag Magnitude

To increase the likelihood of detecting real design errors by the generated vectors, it is desirable to maximize the tag magnitude for which the vector propagates the tag. Ideally, we are interested in finding a vector, if such a vector exists, which can propagate the tag independent of its magnitude. This is important because we do not know anything about the nature of the error and cannot make any assumption for its magnitude.

During the search for finding a solution to the HSAT problem, the HSAT solver fixes the value of variable Δ. This means that the resulting vector will propagate a tag with a specific magnitude to the output. There is no guarantee that a tag with a different magnitude can be detected by the same vector. Consider the following Verilog code:

```
X = 4;
if (X > Y)
     P = 1;
else P = 0;
if (Y == 8)
     P = X;
```

where Y is an input and P is an output. A vector with value 6 for Y will propagate a tag injected in the first line to the output only if the tag magnitude is greater than 2. On the other hand, if the value of Y is set to 8, a tag in the first line can be propagated to the output independent of its magnitude.

The following modifications are made to the HSAT problem in order to maximize the covered magnitude of the tag.

1) Variable substitution is used to eliminate Δ from all equality constraints.
2) All inequalities with Δ present in them, are rewritten in the following form, $\Delta \leq$ linear combination of other variables, or $\Delta \geq$ linear combination of other variables.
3) Δ is replaced by Δ_{ub} and Δ_{lb}, in the first and the second form inequalities, respectively.
4) We maximize $\Delta_{ub} - \Delta_{lb}$ over HSAT constraints.

The result gives us a vector which can propagate the tag, for all values between Δ_{lb} and Δ_{ub}, inclusive. Note that, this algorithm converts the search problem from HSAT feasibility to optimization over HSAT constraints.

An alternative heuristic for maximizing the magnitude of the covered tag is to select paths on the graph which propagates the tags only through operators that can propagate tags independent of their magnitudes. An HSAT problem requiring propagation through such paths is written. This way, only an HSAT feasibility problem needs to be solved. If these constraints make the HSAT problem infeasible, an alternative path must be tried. Consider the example earlier in the section. Since the tag on `X = 4;` can be propagated through the statement `P = X;` independent of its magnitude while `Y` has the value 8, we add the constraint `(Y == 8) = 1` to the HSAT problem.

IV. Experimental Results

In this section we present the experimental results of using our method to generate test vectors for several examples. We also compare different clause orderings.

TABLE I
PERFORMANCE OF THE VECTOR GENERATION APPROACH

Example	#Lines	#Vectors	#Covered Tags	Percent	#Aborted	Random
FIFOctrl	146	15	21	84%	4	56%
DMActrl	443	19	20	16%	102	6%
port	73	13	20	100%	0	85%
counter	100	10	10	59%	7	41%
arbiter	180	54	54	100%	0	100%
crd	191	21	26	54%	22	50%

TABLE II
COMPARING DIFFERENT HEURISTICS FOR ORDERING SAT CLAUSES

Example	#Lines	Random	Topological	Depth-First	Topological and Depth-First
FIFOctrl	146	160 s	159 s	218 s	160 s
DMActrl	443	188 s	142 s	149 s	143 s
port	73	1 s	1 s	1.1 s	1.1 s
counter	100	3 s	2.9 s	3.2 s	3 s
arbiter	180	216 s	114 s	190 s	113 s
crd	191	486 s	408 s	490 s	444 s

A. Performance Comparison

We have implemented the vector generation algorithm proposed in this paper in a prototype system. The implementation uses the VL2MV Verilog parser in VIS verification system [6]. VL2MV converts the Verilog to structural RTL in the BLIF-MV format. Our implementation involved converting the BLIF-MV format to our internal graph representation from which we could generate constraints. In addition, we implemented a coverage computation (tag simulation) routine which operates on the same graph representation. The combination of the linear and Boolean constraints was solved using the HSAT solver system [5]. Each time a vector was generated, it was tag simulated to determine other tags covered by it. We set the upper bound on the number of time frames to 10. The experiments were performed on a Sun Ultra 30/300 with 256 MB of RAM running at 300 MHz.

The examples used in Table I correspond to various circuits from industrial and academic sources implemented in Verilog. **FIFOctrl** is a FIFO controller, **DMActrl** is a DMA controller, **port** is an interface circuit, **counter** is an 8-bit counter, **arbiter** is a bus arbiter, and **crd** is a traffic controller. Note that **counter** and **port** are part of a larger circuit. We used the topological ordering for clauses for this experiment.

The basic numbers highlighting the performance of our vector generation algorithm are presented in Table I. Presented are the number of generated vectors, the number of covered tags, the percentage of the total tags covered by the generated vectors, the number of tags on which the vector generation had to abort, and the coverage percentage achieved by 1000 random vectors.

As one can see from the table, our program was able to achieve full coverage for some examples. In some cases our program was unable to find vectors covering a tag. The reason is that our algorithm targets tags with small depth. Due to completeness of our algorithm (i.e., given a number of time frames and sufficient CPU time it finds a test vector or proves there is no test vector with the desired length), it may not be able to handle large designs. We are currently working on some heuristic vector generation methods for large designs and tags that require a large number of time frames to be detected.

Finally, in some cases the coverage achieved by the random vectors was equal or close to the coverage achieved by our program (e.g., arbiter and crd), while in other cases our program was able to achieve better results using smaller number of test vectors.

B. Clause Ordering Comparison

The purpose of this section is to show the effect of the clause ordering on the vector generation time. In this controlled experiment, tag simulation was disabled so that each heuristic operated on the same set of tags in each example. The results are presented in Table II. Column 3 has the CPU time for the case that clauses are not ordered. Column 4 is the case where clauses are generated in the topological order of their corresponding operators in the graph. Column 5 is the case when the clauses for the operators in the fanout of an injected tag are generated in depth-first search order. Column 6 is for the case when the clauses for the operators in the fanout of an injected tag are generated in the depth-first search order, and clauses for other operators are generated in the topological order. As one can see, the topological order achieves the best results.

REFERENCES

[1] B. Beizer, *Software Testing Techniques*, second ed. New York: Van Nostrand Rheinhold, 1990.
[2] K.-T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *Proc. 30th Design Automation Conf.*, June 1993, pp. 86–91.
[3] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill, "Architecture validation for processors," in *Proc. 22nd Annu. Symp. Computer Architecture*, June 1995, pp. 404–413.
[4] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM: Efficient computation of observability-based code coverage metrics for functional verification," *IEEE Trans. Computer-Aided Design*, vol. 20, pp. 1003–1015, Aug. 2001.
[5] ——, "Functional vector generation for HDL models using linear programming and boolean satisfiability," *IEEE Trans. Computer-Aided Design*, vol. 20, pp. 994–1002, Aug. 2001.
[6] R. K. Braytonothers, "VIS: A system for verification and synthesis," in *Proc. Computer-Aided Verification*, vol. 1102, June 1996, pp. 428–432.