Simulation Vector Generation from HDL Descriptions for Observability-Enhanced Statement Coverage

Farzan Fallah
Fujitsu Labs. of America, Inc.
Sunnyvale, CA

Pranav Ashar
CCRL
NEC USA, Princeton

Srinivas Devadas
Laboratory for Computer Science
MIT, Cambridge

Abstract

Validation of RTL circuits remains the primary bottleneck in improving design turnaround time, and simulation remains the primary methodology for validation. Simulation-based validation has suffered from a disconnect between the metrics used to measure the error coverage of a set of simulation vectors, and the vector generation process. This disconnect has resulted in the simulation of virtually endless streams of vectors which achieve enhanced error coverage only infrequently. Another drawback has been that most error coverage metrics proposed have either been too simplistic or too inefficient to compute. Recently, an effective observability-based statement coverage metric was proposed along with a fast companion procedure for evaluating it.

The contribution of our work is the development of a vector generation procedure targeting the observability-based statement coverage metric. Our method uses repeated coverage computation to minimize the number of vectors generated. For vector generation, we propose a novel technique to set up constraints based on the chosen coverage metric. Once the system of interacting arithmetic and Boolean constraints has been set up, it can be solved using hybrid linear programming and Boolean satisfiability methods. We present heuristics to control the size of the constraint system that needs to be solved. We present experimental results which show the viability of automatically generating vectors using our approach for industrial RTL circuits. We envision our system being used during the design process, as well as during post-design debugging.

1 Introduction

Simulation is by far the primary design validation methodology in IC design and it is likely to remain so for the foreseeable future, especially in the validation of RTL circuits. A reason for this is that the typical RTL circuit is derived from a heterogeneous, *ad hoc* description of the behavior, i.e., there is no formal model of the behavior for the RTL to be compared against. In fact, even if there were such a formal model, verification tools available today are generally not robust enough to perform an automatic, formal comparison of the RTL against the behavioral model. That leaves some form of simulation as the only alternative to compare the I/O response of the RTL against the specification.

Simulation is the most time-consuming task in the design of microchips, and simulation time clearly influences the time-to-market. Vector generation (usually done manually by hordes of "verification engineers" poring over the HDL code) and actual simulation time both contribute to the time spent in validating the design. Clearly, there is a need for manual generation of simulation vectors to check functionality which cannot possibly be covered by any coverage metric. It is our belief, though, that most functional errors can be detected through the prudent use of a compact set of simulation vectors derived from a suite of quality coverage metrics. The choice of coverage metrics will influence the computational requirements of automatic vector generation, and the size of the generated vector set. However, the most important step here is to help designers to develop sufficient confidence in the vectors generated in this manner to cut back on their manual- and random-vector generation efforts. If one could convince designers that automatic vector generators produce a majority of the vectors that verification engineers manually create, that would be a giant step forward, and would result in significant reduction of design turnaround time.

The key question is what constitutes a reasonable suite of coverage metrics? Clearly, in order to detect an error, simulation must visit the site of the error, and subsequently propagate the effect of the error to an observable[1] output. A coverage metric must begin with some error model and impose observability and controllability constraints on the simulation to be useful. An additional requirement is efficient evaluation of the coverage metric, if it is to be used in the inner loop of vector generation. It is obvious that the coverage of all paths in the HDL code or the coverage of all transitions in an FSM model of the implementation [1] results in way too many vectors to be usable. Approaches that generate vectors which cover selected transitions in the FSM model or selected paths in the HDL code are much more practical. Coverage of all statements in the HDL code [2] is more tractable than path or transition coverage, but again not very meaningful since it does not address the observability requirement.

A recent development in coverage metrics for simulation-based validation has been the proposal of an effective observability-based statement coverage metric along with a fast companion procedure for evaluating it [3, 4]. This metric is more accurate than just statement coverage since it also incorporates observability criteria. On the other hand, it is also more practical than path or transition coverage since it leads to a much smaller number of vectors. The evaluation procedure proposed in [4] is computationally very efficient, more so than other approaches. A feature of this coverage metric is that it is pessimistic, making it more likely that a vector generated based on this metric will in fact uncover a real design error. It is our belief that this coverage metric must be a part of any chosen suite of coverage metrics used for vector generation.

1.1 Our Contribution

The contribution of our work is the proposal of a vector generation procedure targeting the observability-based statement coverage metric. We use repeated coverage computation to minimize the number of vectors generated. For vector generation, we propose novel techniques to set up constraints based on our chosen coverage metric. To solve the system of interacting arithmetic and Boolean constraints, we augment recent algorithms for solving hybrid satisfia-

[1] By observable we mean that a monitor or the designer should be able to distinguish between correct and incorrect responses at that location.

bility problems [5]. We present heuristics to control the number of the constraints generated. Analogous to a fault-list in test generation, we maintain a "tag-list" during vector generation. Tags are associated with each variable assignment in the HDL code. After the generation of each vector, coverage by this vector of all tags in the HDL code is determined using the efficient coverage computation procedure from [4]. The corresponding tags are deleted from the tag-list. The process is continued until all tags under our metric are covered, i.e., the tag-list is empty.

A review of the observability-based statement coverage metric along with the coverage computation procedure is provided in Section 2. A review of an algorithm for solving interacting arithmetic and Boolean constraints is provided in Section 3. Details of our vector generation algorithm are in Section 4. Section 5 provides techniques to enhance the basic vector generation algorithm. Section 6 provides examples of the application of our approach on a few industrial examples. Section 7 concludes the paper and describes ongoing work.

2 Observability-Based Statement Coverage and its Computation

The purpose of this coverage metric is to evaluate vectors for their ability to propagate an error at a specific location in the HDL code to some output. This is much harder to do than computing a controllability metric which just determines if a vector results in a statement being visited. A review of previous work on controllability and observability-based metrics is provided in [4].

In [3], the notion of a *tag* to model the possibility that an incorrect value is computed at a location was introduced, where a location corresponds to an assigned variable in some statement in the HDL code. As the name suggests, a tag is just a label. It does not have a value, and has nothing to do with how the erroneous value was generated. It is either present or absent. It does have a sign associated with it which helps determine if it can be propagated forward. As will become clear, these limited attributes associated with the tag lead to an inherent pessimism in the coverage metric. At the expense of a few extra vectors, this pessimism leads to greater confidence in the ability of vectors satisfying the coverage metric to detect real design errors.

Given a vector and a location in the HDL code, a tag at the location is said to be covered by the vector under this coverage metric if it is determined that the tag can be propagated to some output by the vector. The propagation of a tag may be blocked because of interaction between data values. For example, if one input to a two-input multiplier has a tag on it while the other input is zero, the tag is not propagated to the output of the multiplier. Successful tag propagation implies that using the vector for simulation will reveal the error in RTL model. Given a vector, the task of the coverage computation procedure is to efficiently determine all the tags that will be covered by a vector. We use a *single tag* model in which a tag is injected only on one location at a time.

Coverage computation is done by a concurrent algorithm on a graph extracted from the HDL model. Details of the tag propagation algorithm can be found in [4].

3 Solving Hybrid LP-SAT Constraints

Given a Boolean equation, Boolean satisfiability (SAT) is the problem of finding an assignment of variables so that the equation evaluates to one, or to establish that no such assignment exists. Typical SAT algorithms convert the Boolean equation into a conjunctive normal form (CNF) and apply a branch-and-bound algorithm to find the solution, or to prove that none exists. SAT algorithms have found successful application in CAD in stuck-at fault test generation [6].

Vector generation from HDL code targeting a chosen coverage metric can also, in theory, be set up as a purely Boolean satisfiability problem. Given that HDL code consists of word-level arithmetic operators in addition to logic gates, this would not be a very efficient approach in practice. It was demonstrated in [5] that it is much more efficient to keep the Boolean and word-level domains distinct, and only model their interaction for the specific variables shared between the two domains. Linear constraints are solved in the linear programming domain, while Boolean clauses are solved using a SAT solver. Completeness is ensured by effectively ensuring that the feasibility of the linear constraints is checked for each path to a leaf in the branch-and-bound tree of the SAT solver. This is accomplished in practice by checking the feasibility of the linear constraints each time a Boolean variable is assigned in the SAT solver. When there is a choice of the next variable to assign, either a Boolean variable, or a word-level variable with no direct correlation to a Boolean variable is picked. If a word-level variable is picked, the feasibility of a solution must be checked for the entire range of the variable. The feasibility of linear constraints is checked by relaxing the constraint that the word-level variables be integral. If the problem is infeasible with this relaxation, the original problem is infeasible. The integral constraint is imposed only at the leaves of the SAT branch-and-bound tree. Because of the correlation between the Boolean and word-level variables, some optimizations in the SAT solver algorithm cannot be used in the hybrid problem. Even so, it was shown in [5] that the hybrid algorithm is much faster than alternatives.

The problem of solving for a mixture of Boolean and linear constraints was called the Hybrid SAT or HSAT problem in [5]. This is also the approach we will follow in our effort to generate vectors for the observability-based code coverage metric. Details of constraint generation and constraint solution can be found in [5].

4 Vector Generation Algorithm for Observability-Enhanced Statement Coverage

We provide details of our vector generation algorithm in this section. The goal of the algorithm is to generate vectors so that each tag (denoted by Δ) is propagated to an output by some vector.

As the first step, the HDL description of the design is compiled into structural RTL. In our prototype implementation, this involved compiling a Verilog [7] description into BLIF-MV [8]. A graph $G(V, E)$ encapsulating the dependencies between operators is built from the RTL description. There are the following two types of nodes in the graph:

1. Operator nodes, denoted by V_{op}, which correspond to the instantiation of operators in the RTL description.

2. Latch nodes, denoted by V_l, which correspond to latches in the RTL description.

Every edge E corresponds to a variable in RTL description. An edge from V_1 to V_2 exists if there is a data dependence between operators or latches corresponding V_1 and V_2.

4.1 The Basic Algorithm

The basic algorithm operates according to the following steps:

1. A tag-list is set up. This is analogous to the fault-list in stuck-fault test generation. As the algorithm proceeds forward, tags are removed from the list as vectors are found to cover them. Ideally, the tag-list should be empty when the algorithm completes.

2. An upper bound on number of time frames, t_{max}, that will be used for vector generation is selected.

3. If there is no uncovered tag, the algorithm stops. Otherwise it selects an uncovered tag to generate a vector for. The variable V_f and the operator V_{op_f} corresponding to this tag are identified in the graph $G(V, E)$.

4. t denotes the number of time frames that the design will be expanded to in the current attempt. t is set to one.

5. The graph $G(V, E)$ is unrolled t times. V_f and all variables in its transitive fanout are marked.

6. HSAT constraints for both tagged and untagged versions of the circuit are generated according to the techniques in [5]. For the tagged circuit, we ignore constraints with no marked variables in them, and replace V_f by $V_f + \Delta$ in constraints corresponding to V_{op_f}. For the untagged circuit, constraints are only generated for the portion of the circuit in the fanin of marked outputs.

7. Constraints are added expressing the requirement that the tag be detected on at least one of the marked variables which is also an output of the circuit. For example, if variables E_{O1} and E_{O2} are marked outputs, the added observability constraint will be the following,

$$(E_{O1} > E_{O1_f}) \vee (E_{O1} < E_{O1_f}) \vee$$
$$(E_{O2} > E_{O2_f}) \vee (E_{O2} < E_{O2_f}).$$

8. The HSAT problem is solved using the algorithm described in [5], with enhancements as described in Section 5.

9. If there is no solution to HSAT problem and if $t < t_{max}$, t is incremented by 1 and the algorithm reverts to Step 5.

10. If there is no solution to HSAT problem and $t = t_{max}$, the algorithm returns reporting that the tag cannot be covered within t_{max} time frames.

11. For the vector generated, tag simulation is performed using an algorithm akin to [4] to detect all the tags which can be covered by this vector. The algorithm updates the tag-list and reverts to Step 3.

4.2 Example Application of the Basic Algorithm

Consider the following Verilog code as an example to illustrate the above algorithm. The code computes the running sum, `sum`, of the inputs `in`. The output is set equal to the sum if two values of `in` ≥ 8 are observed. Clearly, if we want to detect a tag on the statement `sum = sum + in;`, we must wait for at least one cycle for it to propagate to the output `out`. The RTL circuit generated from this code is shown in Figure 1.

```
module test(clk, in, out)
input  clk;
input  [3:0] in;
output out;
reg    out;
reg    [1:0] i;
reg    [3:0] sum;

initial
 begin
   out = 0;
   i = 0;
   sum = 0;
 end

always(@ posedge clk)
 begin
    if( in >= 8 )
      i = i + 1;

    sum = sum + in;
    if( i == 2 )
        begin
          i = 0;
          out = sum;
        end
    else
        out = 0;
 end
endmodule
```

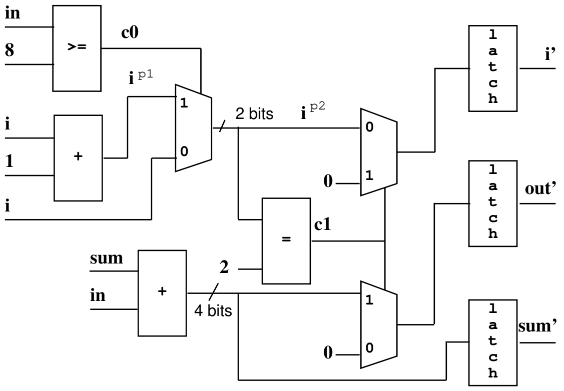

Figure 1: Structural RTL for the Example Verilog Code

The following constraints are generated for the various statements (indicated in comments below) and variables in the untagged version of one unrolling of the RTL circuit. The subscripts $t0$ and $t1$ indicate the time frame to which the variable belongs. The variables i^{p1} and i^{p2} are indicated in Figure 1. $c1^{p1}$ and $c1^{p2}$ are temporary variables.

// initialization
$out_{t0} = 0$
$i_{t0} = 0$
$sum_{t0} = 0$

// in >= 8
$in_{t0} - 8 + 16(1 - c0_{t0}) >= 0$
$in_{t0} - 8 - 16c0_{t0} <= -1$

// if/else (in >= 8)
$i^{p1}_{t0} - i_{t0} = 1$
$i^{p1}_{t0} - i_{t0} = 1$

$i^{p2}_{t0} - i^{p1}_{t0} + 16(1 - c0_{t0}) >= 0$
$i^{p2}_{t0} - i^{p1}_{t0} - 16(1 - c0_{t0}) <= 0$

$i^{p2}_{t0} - i_{t0} + 16c0_{t0} >= 0$
$i^{p2}_{t0} - i_{t0} - 16c0_{t0} <= 0$

// sum = sum + in;
$sum_{t1} - sum_{t0} - in_{t0} = 0$

// (i == 2)
// i >= 2
$i^{p2}_{t0} - 2 + 4(1 - c1^{p1}_{t0}) >= 0$
$i^{p2}_{t0} - 2 - 4c1^{p1}_{t0} <= -1$

// i <= 2
$2 - i^{p2}_{t0} + 4(1 - c1^{p2}_{t0}) >= 0$
$2 - i^{p2}_{t0} - 4c1^{p2}_{t0} <= -1$

`// (i >= 2) && (i <= 2)`

$(\bar{c1}^{p2}_{t0} + \bar{c1}^{p1}_{t0} + c1_{t0})$
$(c1^{p1}_{t0} + \bar{c1}_{t0})$
$(c1^{p2}_{t0} + \bar{c1}_{t0})$

`// if/else (i == 2)`

$i_{t1} + 4(1 - c1_{t0}) >= 0$
$i_{t1} - 4(1 - c1_{t0}) <= 0$

$i_{t1} - i^{p2}_{t0} + 4c1_{t0} >= 0$
$i_{t1} - i^{p2}_{t0} - 4c1_{t0} <= 0$

$out_{t1} - sum_{t1} + 16(1 - c1_{t0}) >= 0$
$out_{t1} - sum_{t1} - 16(1 - c1_{t0}) <= 0$

$out_{t1} + 16c1_{t0} >= 0$
$out_{t1} - 16c1_{t0} <= 0$

We would like to generate a vector to cover the tag on the statement `sum = sum + in;`. To achieve that, all constraints corresponding to the operations in the fanout of sum are duplicated with each marked variable v in the constraints replaced with $tagged\_v$. The tag is injected by means of the statement `tagged_sum = sum + in + Δ` labeled with the appropriate subscripts for the number of unrollings. Constraints for detecting the tag at the output in the first time frame are the following:

$tagged\_out_{t1} - out_{t1} + 16(1 - g1) >= 1$
$out_{t1} - tagged\_out_{t1} + 16(1 - g2) >= 1$
$g1 + g2 >= 1$

Since the tag cannot be detected in the first time frame, the circuit must be unrolled once to generate a two time frame version. This is achieved by marking variables in unrolled circuit and generating constraints for unrolled circuit. For the variables corresponding to the second time frame, the subscripts $t1$ and $t2$ are used instead of $t0$ and $t1$, respectively.

4.3 Relationship to ATPG for Stuck-Faults

Apart from the fact that the constraints generated in our algorithm are a hybrid of linear and Boolean constraints, this manner of setting up constraints for the tag propagation problem is similar to the way constraints are set up in test pattern generation for stuck-at faults [6]. A key difference has to do with the error magnitude. Either we do not assume any error magnitude as a result of which no constraints are generated to justify a particular value at the site of the tag, or we try to maximize the range of error magnitude for which the error is propagated. This is clarified in the following section. There is no corresponding counterpart requirement in the test pattern generation case. In the test generation case, constraints must be generated to justify a value opposite to the stuck-fault value at the fault site. We do need to ensure that the statement corresponding to the tag location is visited during the course of simulation of the generated vector. That, as well as the observability of the tag is ensured by the constraints generated in our algorithm.

5 Enhancements to the Basic Algorithm

We use various heuristics to improve the run time and the quality of vectors. As an straightforward tactic to reduce the vector generation time, the deterministic vector generation is preceded by a limited random vector generation phase. In this case, the extracted graph can be used to compute the values of the variables and after that concurrent tag propagation can be used as before.

5.1 Improving the Performance of the Algorithm

Unrolling the HDL model increases the size of the HSAT problem rapidly. This can degrade the speed of test vector generation substantially. It can easily be seen that in order to detect the 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 the HSAT problem.

As an example, in Figure 1, in order to have a tag on out', it is necessary to have $c1 = 1$. Adding this constraint to the original HSAT problem can help the HSAT solver 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 exponential growth in the number of 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 smaller run time. This is easily done by modifying the step in the algorithm that marks the transitive fanout variables of the tag location. Under this heuristic, the markings propagate forward only along specific paths. Correspondingly, constraints are added to the HSAT problem which require the tag to propagate forward along the chosen paths. 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.

Note that marking a path or a subset of paths in the circuit can result in a decrease in the total number of variables and constraints.

5.2 Maximizing the Tag Magnitude

In order to increase the likelihood that real design errors will be uncovered by the generated vectors, we would like to maximize the tag magnitude for which the vector covers 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 nature of the error and cannot make any assumption for its magnitude. We achieve this in the following manner.

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 the tag with some specified 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 shown below.

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

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 second form inequalities, respectively.
4. We maximize Δ_{ub} - Δ_{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. The optimization problem is harder to solve.

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 magnitude. An HSAT problem requiring propagation through such paths are 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 the magnitude of the tag if `Y` has the value 8, we add the constraint `(Y == 8) = 1` to the HSAT constraints.

5.3 Undetectable Tags

In some cases there might be tags in the code, which cannot be detected by any vector. For example, a tag on the statement `out = 0` in the Verilog code in Section 4.2 cannot be detected since the statement is basically dead code. We would like to detect them as early as possible to avoid wasting time solving HSAT problems for them.

Obviously, if no output variable has been marked (as in Step 5 of the basic algorithm) in t unrollings, we do not generate constraints for the tag until $t+1$ unrollings. Furthermore if the set of marked latches within t_0 time frames ($t_0 < t$) is equal to the set of marked variables within $t_0 + 1$ time frames, and no output has been marked in t time frames, the tag is undetectable. This gives us an easy method for finding some undetectable tags though not all of them. For example, the tag on Y on Line 2 in the following code is not detectable, but that cannot be identified by our heuristic.

```
Line 1        if((X > 4)&&(X < 3))
Line 2          Y = 1;
Line 3        P = Y;
```

This tag can be determined to be undetectable by solving HSAT constraints for the tag ignoring initial values on latches, and writing constraints for detecting the tag on an output variable or latch. If we fail to find a vector, the tag is undetectable. Otherwise it might be detectable.

5.4 Finding Lower Bounds on the Number of Unrollings Required for Each Tag

As discussed in the Section 5.3, we can detect the minimum number of unrollings required for covering a tag by computing the number of unrollings required to obtain a marked output. It turns out that we can compute lower bounds for all tags concurrently by starting with marked outputs and propagating the markings backward to nodes in their transitive fanin. The backward propagation of the markings is continued beyond the latch boundaries by progressively increasing the number of unrollings. The number of unrollings required for a marking to reach an operation is the minimum number of unrollings that will be required to propagate a tag at that location to an output. As an extension, separate markings could be used for each output. The lower bounds for each tag are used to sort the tags according to the likely level of difficulty in covering the tag, or to determine the potential of a vector generated for a tag to cover other tags.

6 Experimental Results

6.1 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 [8]. 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 operated on the same graph representation. The combination of linear and Boolean constraints was solved using the HSAT solver system [5]. Each time a vector was generated, it was tag simulated to determine the other tags covered by it. We did *not* use random vector generation to get a fair evaluation of the vector generation algorithm. The experiments were performed on a Sun Ultra 30/300 with 256 MB of RAM running at 300 MHz.

The examples used in Table 1 correspond to various circuits from industrial and academic sources implemented in VERILOG. **FIFOctrl** is a FIFO controller, **DMActrl** is a DMA controller, **counter** is an 8-bit counter, **port** is an interface circuit, **arbiter** is a bus arbiter, and **crd** is a traffic controller. Note that **counter** and **port** are part of a larger circuit.

We used topological ordering (see below) for clauses, and enabled tag simulation for this experiment. Random vectors were not used as indicated above.

The basic numbers highlighting the performance of our vector generation algorithm are presented in Table 1. Presented are the number of generated vectors, the number of covered tags, the percentage of total tags covered by the generated vectors, and the number of tags on which the vector generation had to abort.

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. It tries to find a vector within several time frames or tries to prove that it does not exist. We are working on heuristic vector generation methods for tags that require a large number of time frames to be detected.

6.2 Clause Ordering Comparison

The purpose of this section is to show the effect of 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 2. 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 clauses for operators in the fanout of the injected tag are generated in depth-first search order. Column 6 is for the case when clauses for operators in the fanout of injected tag are generated in the depth-first search order, and clauses for other operators are generated in topological order. As one can see, using topological order achieves the best results.

Ordering of clauses in SAT can affect the cpu time. A good ordering will help the HSAT solver in selecting good variables for branching and as a result decrease the CPU time.

In depth-first ordering, we try to keep clauses corresponding to a path from a tag to the outputs/latches together. This way, the HSAT solver will choose variables corresponding to a path in the circuit.

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

Table 1: Performance of the Vector Generation Approach

1	2	3	4	5	6
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

Table 2: Comparing Different Heuristics for Ordering SAT Clauses.

In topological ordering, clauses for an operator appear only after the clauses for all the operators in its transitive fanin. This way the HSAT solver is likely to set values on input variables to an operator early. In turn, this is likely to imply a value on the output of operator.

7 Conclusions and Future Work

Our work is a confluence of recent developments in the computation of an observability-based coverage metric, the solution of systems of hybrid linear and Boolean constraints and novel heuristics for generating vectors for observability-enhanced coverage. We have proposed a method for the generation of simulation vectors from hardware description language (HDL) models targeting observability-based statement coverage which uses a fast coverage computation procedure in the inner loop to minimize the number of vectors that need to be generated to cover all statements. The vector generation itself is done by setting up interacting arithmetic and Boolean constraints, and solving them using hybrid linear programming and Boolean satisfiability methods. Heuristics are presented to control the size of the constraint system that needs to be solved. A key contribution has been the proposal of a technique to maximize the range of the error magnitude for which a vector covers a tag. By targeting an effective coverage metric and by using deterministic vector generation, we automatically generate simulation vectors of high quality.

The development of such a system will allow the designer to use this tool during the design process, and not just in a post-design debugging phase.

There are three directions of work to be explored in improving the current system: changing the order of tags in the tag-list, modifying a test vector to cover more tags, and heuristics for handling deep sequential designs.

In the prototype system, the tags (assignments) are processed in the order of their line numbers. In practice it is much better to generate test vectors that cover many tags early during the test vector generation, because this results in deleting many tags from the tag-list before trying to generate a test vector for them, and decreasing the overall tag generation CPU time. Finding a heuristic for changing the order of tags in the tag-list is one possible direction for the future work.

In some cases HSAT problems for covering different tags are very close to each other. As a result, their solutions will be close to each other too. Finding a method for modifying a test vector to cover some new tags can improve the speed of the test vector generation substantially.

The current prototype uses a complete algorithm to generate test sequences. It is necessary to add heuristics to the current algorithm to handle deep sequential designs.

References

[1] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill, "Architecture Validation for Processors," in *Proceedings of the 22^{nd} Annual Symposium on Computer Architecture*, June 1995.

[2] K.-T. Cheng and A. S. Krishnakumar, "Automatic Functional Test Generation Using the Extended Finite State Machine Model," in *Proceedings of the 30^{th} Design Automation Conference*, pp. 86–91, June 1993.

[3] S. Devadas, A. Ghosh, and K. Keutzer, "An Observability-Based Code Coverage Metric for Functional Simulation," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 418–425, November 1996.

[4] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM: Efficient Computationa of Observability-Based Code Coverage Metrics for Functional Simulation," in *Proceedings of the 35^{th} Design Automation Conference*, pp. 152–157, June 1998.

[5] F. Fallah, S. Devadas, and K. Keutzer, "Functional Test Generation Using Linear Programming and 3-Satisfiability," in *Proceedings of the 35^{th} Design Automation Conference*, pp. 528–533, June 1998.

[6] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design*, vol. 11, pp. 4–15, January 1992.

[7] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Boston, MA, second ed., 1994.

[8] R. K. Brayton and others, "VIS: A System for Verification and Synthesis," in *Proc. Computer-Aided Verification*, vol. 1102, pp. 428–432, June 1996.