

# A Novel Approach for Functional Coverage Measurement in HDL

<sup>1</sup>Chien-Nan Jimmy Liu, Chen-Yi Chang, Jing-Yang Jou, <sup>2</sup>Ming-Chih Lai and Hsing-Ming Juan

<sup>1</sup>Department of Electronics Engineering  
National Chiao Tung University  
HsinChu, Taiwan, R.O.C.

{ jimmy, chenyi, jyjou } @ EDA.ee.nctu.edu.tw

<sup>2</sup>NOVAS Software Inc.

2F, 60 Park Avenue II, Science-Based Ind. Park  
HsinChu, Taiwan, R.O.C.

{ kelvin, ming } @ novasoft.com.tw

## Abstraction

While the coverage-driven functional verification is getting popular, a fast and convenient coverage measurement tool is necessary. In this paper, we propose a novel approach for functional coverage measurement based on the VCD files produced by the simulators. The usage flow of the proposed dumpfile-based coverage analysis is much easier and smoother than that of existing instrumentation-based coverage tools. No pre-processing tool is required and no extra code will be inserted into the source code. Most importantly, the flexibility in choosing coverage metrics and measured code regions is increased. Only one simulation run is needed for any kind of coverage reports. By conducting some experiments on real examples, it shows very promising results in terms of the performance and the accuracy of coverage reports.

## 1. Introduction

Due to the increasing complexity of modern circuit design, verification has become the major bottleneck of the entire design process. Until now, the functional verification is still mostly done by simulation with a massive amount of test patterns. During simulation, an important question is often asked: Are we done yet? In most cases, the quality of the test mainly depends on the designer's understanding of the design and is not measurable. Therefore, more objective methods, which perform a quantitative analysis of simulation completeness [1, 2] with some well-defined functional coverage metrics, are proposed and rapidly getting popular. With the coverage reports, the designers can focus their efforts on the untested areas and greatly reduce the verification time. Although 100% coverage cannot guarantee a 100% error-free design, it provides a more systematic way to gauge the completeness of the verification process.

For this purpose, a lot of functional coverage metrics [3, 4] are proposed to verify the designs written in HDL. Although so many different metrics have been proposed, not a single metric is popularly accepted as the only complete and reliable metric. Therefore, designers often use multiple metrics to evaluate the completeness of a simulation. In order to get those coverage reports, an extra tool is required besides the simulator. Several commercial tools [5, 6, 7] for HDL code coverage are now available. In those approaches, they insert some special PLI (Programming Language Interface) [8] tasks into the source code and re-run the simulation again to obtain the coverage. The typical usage flow of those instrumentation-based tools is shown in Figure 1.

From Figure 1, we can see some inconvenience existing in this usage flow. First, a pre-processing tool should be executed before the simulation to insert PLI tasks. It forces the users to change their original design flow for coverage analysis. Second, a lot of PLI tasks will be inserted into the source code. It incurs high overhead

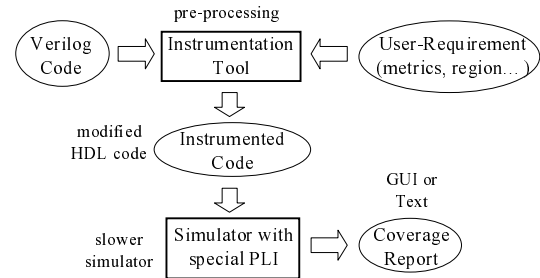


Figure 1 : Usage flow of instrumentation-based tools.

on both code size and simulation performance. Third, because the PLI routines are compiled with the simulator, to run the coverage analysis requires an extra simulation run. Most importantly, they have less flexibility in choosing different coverage metrics and measured code regions. Because the coverage data they can obtain depend on where the PLI tasks are put, the users who want to see another coverage reports for different metrics or different code regions have to re-run the pre-processing tool and the long simulation again.

In the debugging phase, the designers often ask the simulator to generate a value change dump (VCD) file [8] for post-processing such as the waveform viewers. This file records every value change of selected variables during simulation so that we may be able to recover the execution status from it. If the coverage data could be obtained from the dumpfiles, the usage flow of the coverage analysis will be much easier and smoother as shown in Figure 2. No pre-processing tool is required so that the design flow can be kept the same as usual. In addition, no extra code is inserted so that the overhead on code size and simulation time can be eliminated. Most importantly, the flexibility in choosing coverage metrics and measured code regions is increased. Only one simulation run is needed for any kind of coverage reports because all of them can be derived from the same dumpfiles.

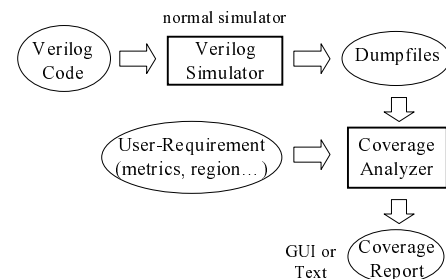


Figure 2 : Usage flow of dumpfile-based analysis.

Based on this observation, we propose a novel approach to measure functional coverage from dumpfiles in this paper. By carefully considering the feasibility, we can handle almost all descriptions accepted by the simulators in this approach. The

experiments on real examples show very promising results in terms of the performance and the accuracy of coverage reports.

The remainder of this paper is organized as follows. First, the HDL model we use in our algorithm is described in Section 2. The operations of our dumpfile-based coverage analysis algorithm are presented in Section 3. In Section 4, we show some experimental results with five real designs. Finally, the conclusions and future works are given in Section 5.

## 2. HDL Modeling

Until now, most HDL simulators are still based on the event-driven approach. Therefore, it is convenient to model the HDL designs as some interacting events. In this work, we extend the *event graph* in [9] to a 3-D event graph, which is a directed graph  $G(V, E)$  with hierarchy, and use it to be the HDL model.

In our model, an event is defined as the largest unit of a HDL program that will be executed atomically during simulation, i.e., the code in an event will be executed during one step of an event-driven simulation. Each vertex  $v \in V$  in the event graph represents an event in the HDL designs with some possible child nodes  $child(v) \in V$ , which are *generated* by  $v$ . In HDL programs, if the code of an event  $j$  is put into another event  $i$  and separated by an `@`, `wait` or delay statement (`#d`), the event  $j$  can be regarded as being generated by the event  $i$  once the event  $i$  is executed. Those child nodes are generated temporarily and will be deleted after executed. Each directed edge  $e(i, j) \in E$ , where  $i$  and  $j \in V$ , represents that the event  $j$  is *triggered* by the event  $i$ . In HDL programs, if there are any variables in the sensitivity list of event  $j$  being the left-hand-side variables of the assignments in the event  $i$ , we say event  $j$  is triggered by the event  $i$ .

In order to explain the event graph more clearly, an example of the event graph for a small Verilog program is shown in Figure 3. The example shown in Figure 3(a) can be partitioned into five events. Their relationship is built in the event graph shown in Figure 3(b). The dashed arrows represent their hierarchical relationship, and the solid ones represent their triggering relationship. Each vertex in the event graph has two rows. The upper row is the *triggering condition* of this event, and the lower row is the executable code of this event. Because the event graph may be changed dynamically, we only show the event graph at  $\$time = 0^+$ . At  $\$time = 0$ , E1 and E2 are executed. Then E11 and E21 are generated respectively at  $\$time = 0^+$  as shown in Figure 3(b). E11 will be executed after a constant delay when E1 is executed, so we set its triggering condition as the absolute time that it will be executed. Because event E22 has not been executed yet at that time, it is not shown in Figure 3(b).

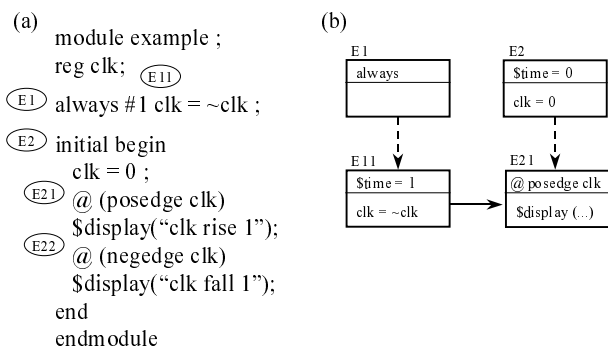


Figure 3 : (a) A Verilog program. (b) The event graph at time  $0^+$ .

In this simple example, we can directly show the executable code of each event in the event graph. However, in real cases, the executable code of an event could be very complex. Therefore, we use the *statement tree* to retain the complex actions of each event. A statement tree is a rooted, directed graph with vertex set  $N$  containing two types of vertices. A *nonterminal* vertex  $n$  has one or more children  $child(n) \in N$ . A *terminal* vertex  $n$ , which represents the terminal block in the HDL codes, has no children but a set of assignments, which are recorded in  $action(n)$ . The *entering conditions* of each vertex  $n$  are recorded in  $cond(n)$ .

## 3. Dumpfile-Based Coverage Analysis

### 3.1 Value Change Dump Files

Value change dump (VCD) files [8] record every value change of selected variables during simulation. With the VCD feature, we can save the value changes of selected variables for any portion of the design hierarchy during any specified time interval. The VCD format is a very simple ASCII format. Besides some simple commands which define the parameters used in the file, the format is basically a table of the values of variables with timing information. Because the VCD files keep all the value information during simulation, they are widely used in many tools for post-processing such as the waveform viewers.

### 3.2 Dumpfile-Based Coverage Analysis

Since the dumpfiles record every value change of selected variables, if we read a dumpfile from the beginning of it, we can obtain the value of each variable recorded in the dumpfile at any specific time. Then we can exactly know which code has been executed and what the execution count is according to the dumpfiles with properly selected variables. By using these statistics, the code coverage can be easily calculated without running the simulation again.

Although the operations required for our DUCA (DUmpfile-based Coverage Analysis) algorithm are very similar to those in a simulation, the complexity is much lower because many operations can be skipped. In coverage analysis, the major concern is to decide which code is executed, not the value of each variable. Therefore, the operations whose results do not change the control flow are skipped. As a result, only a few operations have to be evaluated again in our DUCA algorithm, and the computing complexity can be significantly reduced.

#### 3.2.1 Variable Selection

As described above, not all information is required to decide which part of code is executed. In order to improve the efficiency of our DUCA algorithm, an analysis should be performed first to find the set of variables that affect the decisions. Actually, they are either the variables appearing in the triggering conditions of each vertex in the event graph or the entering condition of each vertex in the statement trees. This variable selection operation also provides the capability of partial coverage analysis. If the user's concern is only the code coverage of a part of the entire code, we can select only those variables required for this part of code so that the analysis time can be further reduced.

#### 3.2.2 Running Analysis

After conducting the variable selection step, we can start coverage analysis by tracing the changes of these signals in the dumpfiles. In order to explain the trace operation more clearly, an example is shown in Figure 4. Given a value change of a signal in

the dumpfiles, we first find the fanouts of this signal, which are the code affected by this change, and then put marks on the associated positions in the event graph and the statement trees. If the affected code is the triggering condition of a vertex in the event graph, we mark it as “Triggered”; if the affected code is the entering condition of a vertex in the statement trees, we mark it as “Modified”. With those marks, we can avoid a lot of duplicate computations.

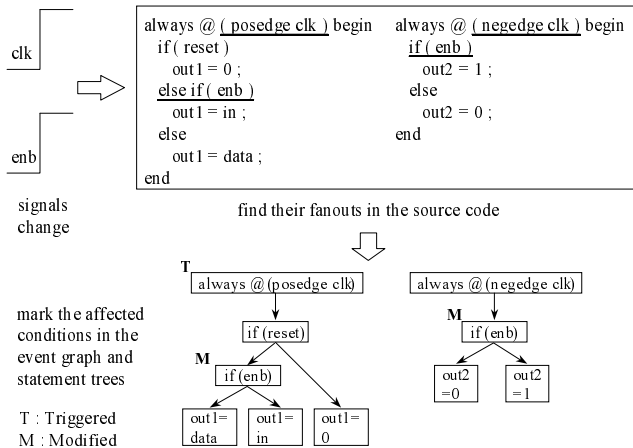


Figure 4 : An illustration of the signal change tracing.

After all the signal changes at the same time are processed, we can traverse the statement trees of the triggered events to decide which code is executed. While traversing down the statement trees, if the current vertex is marked as modified, the result of its entering condition may have been changed and has to be evaluated again to decide which path to go; if not, we can use the previous result to eliminate unnecessary computing. Then the used marks should be cleared for the use of next labeling process. The condition evaluation of the vertices that are marked as modified but not in the traversing paths is also skipped because it is unnecessary now. The “Modified” labels of these vertices are kept until they are finally in the traversing paths.

### 3.2.3 Concurrent Events

Typically, there are a lot of concurrent events in a HDL programs. In event driven simulators, two events may occur at the same simulation time unit with a “delta delay” apart. They will be executed sequentially even though they should occur at the same time unit. However, the “delta delay” is ignored while recorded in the dumpfile. If we cannot recover this sequential relationship, we may use the wrong values while referencing the values of variables.

For this purpose, we keep the triggering information in the edges of the event graph. If an edge exists between two events, they must have a delta delay in between. Then we can easily recover the execution sequence between events by performing a bread-first search (BFS) on the subgraph composing of triggered events. The value of each variable in an event will be updated after the evaluation of this event is completed. In other words, all value references will take the value just before evaluating this event. It makes the later events take the updated value even if they are in the same simulation time unit but with delta delays.

### 3.2.4 Non-Blocking Assignments

In HDL programs, while a non-blocking assignment ( $=>$ ) is executed, its effect does not appear until the end of this time unit. That means all the blocking assignment ( $=$ ) will be executed prior

to the non-blocking assignments even they are located after the non-blocking assignments in the HDL code. It can be viewed as if the simulator divides one time unit into two sequential parts. The blocking assignments are in the first part, and the non-blocking assignments are in the second part. Therefore, we use a two-pass traversal to traverse the event graph. The first traversal only handles the blocking assignments, and the non-blocking assignments are handled in the second traversal. Then all the issues introduced by non-blocking assignments can be resolved.

### 3.2.5 Wait Statements

If there are *wait* or *@* statements in the HDL program, some events will appear as embedded in another event. Taking the Verilog program shown in Figure 3 as an example, event E21 is embedded in the statement tree of event E2 in the initial event graph. In other words, event E2 has no child node at the beginning. Once event E2 is triggered and the embedded event E21 is found in the traversing path of its statement tree, event E21 is generated as a child node of event E2 as shown in Figure 3(b). This kind of events is generated temporarily and will be deleted after executed. As event E21 appears, event E2 cannot be executed until event E21 is completed. Therefore, an extra step should be performed to check the existence of child nodes while searching for the triggered events in the event graph. If child nodes exist, this event cannot be executed until all the child events are completed.

### 3.2.6 Coverage Report

After traversing the dumpfiles, we can obtain the statistics on which code is executed and what the execution count is. However, there are many different coverage metrics which require different coverage reports. Therefore, a post-processing step is added to generate the reports for the user-specified coverage metrics. Most of the coverage reports can be easily obtained from those statistics with little computation overhead. This feature provides the capability of switching the report for different coverage metrics easily. The users who want to see the coverage report of another coverage metric with the same input patterns do not have to re-insert PLI tasks and re-run the long simulation again.

### 3.2.7 DUCA Algorithm

In order to give a summary of the above techniques, the overall pseudo code of our DUCA algorithm is shown in Figure 5.

```

DUCA (hdl_code H, dump_file D, selected_coverage C) {
// analyze the source code and build the event graph
Event_Graph ← CodeAnalysis(H);

// traversing the dumpfile
for each time change t in D {
// find executed codes by the event graph
Stats ← ExecutedCode(Event_Graph, t);
for each signal change s at time t in D {
// mark nodes in the event graph and stmt trees
MarkTree(Event_Graph, s, t);
}
}

// generate coverage report for selected coverage
GenReport(Stats, C);
}

```

Figure 5 : The pseudo code of DUCA algorithm.

## 4. Experimental Results

According to the proposed algorithm, we implement a prototype *DUCA* in C++ language. To conduct experiments, we applied *DUCA* to five real designs written in Verilog HDL. The design statistics are given in Table 1. The number of lines and the number of basic blocks [4] in the original HDL code are given in the columns *lines* and *blocks* respectively. The number of nodes in the event graph is given in the column *events*. The design *ISDN-L2* is an ISDN level-2 receiver. The design *PCPU* is a simple 32-bit DLX CPU with 5-stage pipeline. The design *Divider* is an 8/8 divider with input and output latches. The design *MEP* is a block matching processor for motion estimation. The design *MPC* is a programmable MPEG-II system controller.

Design	lines	blocks	events
ISDN-L2	630	156	48
PCPU	990	155	64
Divider	755	72	80
MEP	2105	217	116
MPC	4130	632	140

Table 1 : Design Statistics.

The experimental results of *DUCA*, which are obtained on a 300MHz UltraSparc II with the simulator Verilog-XL 2.5.16, are shown in Table 2 and Table 3. In Table 2, we demonstrate the performance of *DUCA*. The column *vectors* gives the number of random vectors used for simulation. The CPU time required for pure simulation and coverage analysis are given in the columns *simulation* and *analysis* respectively. The gained speedup ratio, which is defined as (*simulation* / *analysis*), is given in the column *speedup*. In Table 3, we show three kinds of measured coverage data and the accuracy of those results. Because random patterns are used in those experiments, the coverage may not very good. But it does not affect the accuracy of our tool.

As shown in Table 2, our *DUCA* takes less time than simulation on average. In other words, we could have better performance than instrumentation-based tools because an extra simulation run is often necessary in those tools so that the speedup ratio is surely smaller than 1. For the larger cases, we can obtain a good speedup ratio because a lot of operations can

Design	vectors	simulation	analysis	speedup
ISDN-L2	130560	30.40 s	35.05 s	0.87
PCPU	130560	26.48 s	28.47 s	0.93
Divider	130560	25.18 s	12.26 s	2.05
MEP	130560	57.38 s	40.47 s	1.42
MPC	130560	60.68 s	45.07 s	1.35
average				1.32

Table 2 : Experimental results of *DUCA*.

Design	Statement Coverage	Decision Coverage	Event Coverage	Accuracy
ISDN-L2	98 %	73 %	100 %	100 %
PCPU	59 %	60 %	100 %	100 %
Divider	91 %	92 %	100 %	100 %
MEP	99 %	42 %	100 %	100 %
MPC	99 %	48 %	100 %	100 %

Table 3 : The coverage data obtained in the experiments.

be skipped in the coverage analysis. For the smaller cases, the number of operations required in simulation is relatively small. Therefore, the number of skipped operations is also small such that the speedup is not obvious. In addition, the Verilog simulator is a well-developed commercial tool which has been greatly optimized for speed. If our prototype can be carefully optimized by experienced engineers, we think the speedup will be better.

## 5. Conclusions and Future Works

In this paper, we proposed a novel approach for functional coverage measurement in HDL. The usage flow of the proposed dumpfile-based coverage analysis can be much easier and smoother than that of existing instrumentation-based coverage tools. No pre-processing tool is required in our usage flow so that the design flow can be kept the same as usual. In addition, no extra code is inserted such that the overhead on code size and simulator performance can be eliminated. Most importantly, the flexibility in choosing coverage metrics and measured code regions is increased. Only one simulation run is needed for any kind of coverage reports because all of them can be derived from the same dumpfiles. The experiments on real examples show very promising results.

In this paper, the proposed algorithm is designed for measuring the execution-based coverage metrics such as the statement coverage and the decision coverage. However, it does not mean that the value-based coverage metrics such as the FSM coverage and the toggle coverage cannot be measured. Actually, those coverage metrics can be obtained from the same dumpfiles in much easier ways. Because the dumpfiles record the values of variables, if we retrieve the values of the desired variables, such as the state variables of a FSM, from the dumpfiles, we can easily obtain those coverage data. Therefore, measuring coverage from the dumpfiles could be a practical solution for most existing coverage metrics.

Although this algorithm can handle almost all descriptions accepted by the simulator, it may still fail when there is not enough information in the dumpfiles. The designers often dump all information into the dumpfiles in the debugging phase, but they often dump only portions of variables in the integration phase to save simulation time. In this situation, we need another techniques to recover the other required information from existing data such that the coverage analysis can still be performed. Developing such techniques and integrating those techniques into the proposed coverage analysis algorithm is a subject of our future works.

## References

- [1] A. Gupta, S. Malik, and P. Ashar, "Toward Formalizing a Validation Methodology Using Simulation Coverage", 34<sup>th</sup> DAC, Jun. 1997.
- [2] Mike Benjamin, Daniel Geist, Alan Hartman, Yaron Wolfsthal, Gerard Mas, and Ralph Smeets, "A Study in Coverage-Driven Test Generation", 36<sup>th</sup> Design Automation Conference, Jun. 1999.
- [3] Tsu-Hwa Wang and Chong Guan Tan, "Practical Code Coverage for Verilog", Int'l Verilog HDL Conference, Mar. 1995.
- [4] D. Drako and P. Cohen, "HDL Verification Coverage", Integrated Systems Design Magazine, June 1998. (<http://www.isdmag.com/Editorial/1998/CodeCoverage9806.html>)
- [5] CoverMeter™, Advanced Technology Center. (<http://www.covermeter.com>)
- [6] CoverScan™, Design Acceleration Incorporation. (<http://www.designacc.com/products/coverscan/index.html>)
- [7] HDLScore™, Summit Design Incorporation. (<http://www.summit-design.com/products/hdlscore.html>)
- [8] Cadence Reference Manuals.
- [9] R. S. French, M. S. Lam, J. R. Levitt, and K. Olukotun, "A General Method for Compiling Event-Driven Simulations", 32<sup>nd</sup> DAC, 1995.