1 Introduction

With the increase in size and complexity of modern integrated circuits, it has become imperative to address critical verification issues in the design cycle. The process of verifying correctness of designs consumes between 60% and 80% of design effort [1]. For every designer the number of verification engineers can vary from 2 to 4 depending on the design complexity. Moreover, validation is so complex that, even though it consumes the most computational resources and time, it is still the weakest link in the design process. Ensuring functional correctness is the most difficult part of designing a hardware system [2].

In order to verify the correctness of a design, different test cases are generated. Due to the fact that it is impractical to verify exhaustively all possible inputs and states of a design, the confidence level regarding the quality of the design must be quantified to control the verification effort. The fundamental question is: How do I know if I have verified or simulated enough? Verification coverage is a measure of confidence and it is expressed as a percentage of items verified out of all possible items. Different definitions of items give rise to different coverage measures or coverage metrics.

Various coverage metrics exist such as code coverage, parameter coverage and functional coverage. In this paper, only code coverage would be used, which provides insight into how thoroughly the code of a design is exercised by a suite of simulations. The main disadvantage of code coverage metrics lies in the fact that they only measure the quality of the test case in stimulating the implementation and do not necessarily prove its correctness with respect to the specification. On the other hand, code coverage analysis is a well-defined, well-scalable procedure and, thus, applicable to large designs. The goal of current work is to propose a method for speeding up the analysis by implementing new models for simulating coverage items.

The first published reference about code coverage was as early as in 1963 by Miller and Maloney in [3]. Over the following years a large variety of code coverage metrics have been proposed, including statement coverage, block coverage, path coverage, branch coverage, expression coverage, transition coverage, sequence coverage, toggle coverage etc [2][4]. The statement coverage metric measures the number of times every instruction is exercised by the program stimuli. Toggle coverage shows whether and how many times nodes in the design toggle, i.e. how many bits change their state from 0 to 1 or vice versa. In the case of branch coverage, we measure the number of times each branch in the control flow graph of the code is taken or not taken under the set of program stimuli. Path coverage measures the number of times every path in the control flow graph is exercised by the set of program stimuli. A potential goal of software testing is to have 100% path coverage, which implies branch and line coverage. However, full path coverage is a very stringent requirement as the number of paths in a program may be exponentially related to program size.

In this paper, we present a method and a tool for fast analysis of classical code coverage metrics, such as statement, branch and toggle coverage. We introduce High-Level Decision Diagrams (HLDD) model for efficient code coverage analysis and show how those classical coverage metrics map to HLDD constructs. We show that HLDDs can be seamlessly applied to observability coverage analysis, thus, replacing the classical D-calculus based methods (e.g. [13]). Current work is motivated by our previous encouraging research results obtained on HLDD based simulation [5, 6] and test pattern generation [7]. This is the first attempt to use HLDD models in validation and code coverage analysis.

The paper is organized as follows. Section 2 defines the HLDD based graph model for which different coverage metrics were built in. Section 3 shows how HLDDs can be used for measuring code coverage including observability coverage. Section 4 presents comparison with a popular simulation tool for hardware description languages. Finally, Section 5 concludes the paper.
2 High-Level Decision Diagrams

Decision Diagrams (DD) have been used in verification for about two decades. Reduced Ordered Binary Decision Diagrams (BDD) [8] as canonical forms of Boolean functions have their application in equivalence checking and in symbolic model checking. Recently, a higher abstraction level DD representation, called Assignment Decision Diagrams (ADD) [9], have been successfully applied to, both, register-transfer level (RTL) verification and test [10, 11].

The main issue with the BDDs and assignment decision diagrams is the fact that they allow logic or RTL modeling, respectively. In this paper we consider a different diagram representation, High-Level Decision Diagams (HLDD) that, unlike ADDs can be viewed as a generalization of BDDs. HLDDs can be used for representing different abstraction levels from RTL to behavioral. It has proven to be an efficient model for simulation and fault modeling since it provides for a fast evaluation by graph traversal and for easy identification of cause-effect relationships [5, 6].

2.1 Basic definitions

Definition: A HLDD representing a discrete function \( y=f(x) \) is a directed acyclic labeled graph that can be defined as a quadruple \( G=(M,E,X,D) \), where \( M \) is a finite set of vertices (referred to as nodes), \( E \) is a finite set of edges, \( X \) is a function which defines the variables labeling the nodes and the variable domains, and \( D \) is a function on \( E \). The function \( X(m_i) \) returns the variable letter \( x_i \), which is labeling node \( m_i \). Each node of a HLDD is labeled by a variable. In special cases, nodes can be labeled by constants or algebraic expressions. An edge \( e \in E \) of a HLDD is an ordered pair \( e=(m_i,m_j) \in E \), where \( E \) is the set of all the possible ordered pairs in set \( E \). \( D(e) \) is a function on \( E \) representing the activating conditions of the edges for the simulating procedures. The value of \( D(e) \) is a subset of the domain of the variable \( x_i \) denoted by \( X_i \), where \( e=(m_i,m_j) \). It is required that \( Pm_i=\{D(e) \mid e=(m_i,m_j) \in E \} \) is a partition of the set \( X_i \). HLDD has only one starting node (root node), which is a terminal node.

Fig. 1 presents an example of a graphical interpretation of an HLDD.

2.2 Modeling digital systems by HLDDs

In HLDD models representing digital systems, the non-terminal nodes correspond to conditions or to control signals, and the terminal nodes represent operations (functional units). Register transfers and constant assignments are treated as special cases of operations. When representing systems by decision diagram models, in general case, a network of HLDDs rather than a single HLDD is required. During the simulation in HLDD systems, the values of some variables labeling the nodes of a HLDD are calculated by other HLDDs of the system. Fig. 2 presents an example of an HLDD for two variables, state and RMAX in the ITC99 benchmark b04.

3. Code Coverage Analysis on HLDDs

3.1 Simulation at RTL and behavioral levels

The basis for code coverage analysis in this paper is a simulator engine relying on HLDD models. We have implemented an algorithm supporting, both, Register-Transfer Level (RTL) and behavioral design abstraction levels. In the RTL style, the algorithm takes the previous time step value of variable \( x_i \) labeling a node \( m_i \), if \( x_i \) represents a clocked variable in the corresponding HDL. Otherwise, the present value of \( x_i \) will be used.

In the case of behavioral HDL coding style HLDDs are generated and ranked in a specific order to ensure causality. For variables \( x_i \) labeling HLDD nodes the previous time step value is used if the HLDD diagram calculating \( x_i \) is ranked after current decision diagram. Otherwise, the present time step value will be used.

Figure 3. HLDD for an RTL datapath
Algorithm 1 presents the HLDD based simulation engine for RTL, behavioral and mixed HDL description styles.

**Algorithm 1.** RTL/behavioral simulation on HLDDs

For each diagram G in the model

```plaintext
mCurrent = m0;

Let xCurrent be the variable labeling mCurrent

While mCurrent is not a terminal node

If is xCurrent clocked or its DD is ranked after G then

Value = previous time-step value of xCurrent

Else

Value = present time-step value of xCurrent

End if

If Value ∈ D(eactive), eactive = (mCurrent, mNext) then

mCurrent = mNext

End if

End while

Assign xCurrent to the DD variable xG

End for
```

### 3.2 Advantages of HLDD modeling

As an example of modeling systems by HLDDs, consider a subnetwork of a digital system and its HLDD depicted in Figure 3. Here, $R_1$ and $R_2$ are registers ($R_2$ is also a primary output), $M_1$, $M_2$ and $M_3$ are multiplexers, $+$ and $*$ denote adder and multiplier, $I$N$ is an input bus, $y_1$, $y_2$, $y_3$ and $y_4$ serve as input control variables, and $a$, $b$, $c$, $d$, $e$ denote internal buses. In the HLDD, the control variables $y_1$, $y_2$, $y_3$ and $y_4$ are labeling internal decision nodes of the HLDD with their values shown at edges. The terminal nodes are labeled by constant $\emptyset$ (reset of $R_2$), by word variables $R_1$ and $R_2$ (data transfers to $R_2$), and by expressions related to data manipulation operations of the network. By bold lines and colored nodes, a full activated path in the HLDD is shown from $X(m^0)=y_4$ to a terminal node $X(m^T)=R_1*R_2$, which corresponds to the pattern $y_4=2$, $y_3=3$, and $y_2=0$. The part of the network that is activated by this pattern is denoted by colored boxes.

The main advantage and motivation of using high-level DDs compared to the netlists of primitive functions are the increased efficiency of simulation and diagnostic modeling because of direct and compact presentation of cause-effect relationships. For example, instead of simulating the control word $y_1$, $y_2$, $y_3$, $y_4$ = \{0, 0, 3, 2\} by computing the functions

$a = R_1$, $b = R_1$, $c = a + R_1$, $d = b * R_2$, $e = d$, and $R_2 = e$.

we need only to trace the nodes $y_4$, $y_3$ and $y_2$ on the HLDD and compute a single operation $R_2 = R_1 * R_2$.

### 3.3 Mapping coverage metrics to HLDD

In order to analyze quality of verification of hardware designs translated to HLDDs three traditional coverage metrics were chosen and built in to the HLDD based simulation tool. These include statement coverage, branch coverage and toggle coverage. As it was mentioned above, the statement coverage measures the number of times every instruction is exercised by the program stimuli. Toggle coverage shows whether and how many times nodes in the design toggle, i.e. how many bits change their state from 0 to 1 or vice versa. In the case of branch coverage, we measure the number of times each branch in the control flow graph of the code is taken or not taken under the set of program stimuli.

![Fig. 2. b04 example: HLDDs for variables state and RMAX](image-url)
The statement coverage maps directly to the ratio of nodes \( n_{\text{current}} \) traversed during the HLDD simulation presented in Algorithm 1. For example, see Fig. 2 for HLDD representations of state and data register variables in a VHDL design. Covering all nodes in the HLDD model corresponds to covering all statements in the respective HDL. However, the opposite is not true. HLDD node coverage is slightly more stringent that HDL statement coverage. This is due to the fact that in HLDDs diagrams are generated to each data variable separately. Such partition on variables includes an additional context to statement coverage.

Similar to the statement coverage, branch coverage has also very clear representation in HLDD simulation. The ratio of every edge \( e_{\text{active}} \) activated in the simulation process of Algorithm 1 constitutes to HLDD branch coverage.

HLDD toggle coverage is calculated similarly to traditional HDL toggle coverage. However, in this paper a more stringent approach has been selected, where, both, rising and falling front toggling are counted separately. Furthermore, toggling is measured in HLDD nodes, which outnumber the HDL variables.

For example, the branch coverage item corresponding to DATA_IN > RMAX = true in the VHDL code of the b04 front toggling are counted separately. Such partition on variables includes an additional context to statement coverage.

3.4 Observation coverage on HLDDs

Keeping track of covered lines of code does not generally reflect if the respective items influence the primary outputs of the system. The quality of validation is low when only code coverage items corresponding to the internal lines of the system are exercised but not propagated to the system outputs. Furthermore, while the general function of the system is specified at the outputs the internal signals may be difficult for the designer or verification engineer to comprehend and verify.

Fallah et al. [13] propose observation coverage in their method called OCCOM, where simplified fault grading is carried out in order to assess, which code items have been covered and propagated to an observable output. They show that 100% code coverage corresponds to 60-80% observable coverage in the worst case.

However, the OCCOM method [13] and its recent improvements are based on representing the effect of an error by a tag that can propagate through the circuit according to a set of rules similar to the D calculus. The main problem of the method is that it over-simplifies the fault-effect propagation. In this paper, we propose an alternative approach, where a straightforward analysis on HLDDs is used for observability analysis.

Observability analysis on HLDDs is carried out as follows. First, the HLDD \( G \), where the code coverage item has been detected is set to a faulty value by inverting the real simulated value. Then, concurrent fault simulation on HLDDs is carried out. This high-level fault simulation allows analysis of several code coverage items in parallel. The procedure is explained below.

In fault propagation for the digital system \( S = (X,F) \) through a high-level block with a function \( x = f(x_1, x_2, ..., x_m) = f'(x) \), \( X' \subset X \), which is represented by a HLDD \( G_x \), we proceed from the fact that the errors may have been propagated to all of the variables \( x_j \in X' \) used in labels of nodes in the graph. To each node \( m \) of the HLDD with the label \( x(m) \), a complex pattern \( T_{X(m)} = \{P_{X(m),0}, (P_{X(m),1},D'_{X(m),1}), ..., (P_{X(m),k},D'_{X(m),k})\} \) corresponds. From this pattern, it results that a set of error values \( D_{X(m)} = D'_{X(m),0} \cup ... \cup D'_{X(m),k} \) has been propagated to the node \( m \). Let \( D \) be the set of all errors currently activated and listed in \( T_{X(m)} \).

Consider the fault simulation on the HLDD \( G_z \) as the following set of procedures.

Procedure 1. The fault-free path is simulated in accordance to the fault-free input pattern \( P_{X(m),0} \) and the fault-free value of \( x = 0(m^{0}) \) is calculated, where \( m^{0} \) is the terminal node of the fault-free activated path.

Let us denote the set of all nodes traced in the fault-free path up to the node \( m \) (itself not included) by \( M_{P_{X(m),0}}(m) \).

Let \( D_{P_{X(m),0}}(m) \) be the set of all faults propagated to the nodes \( m \in M_{P_{X(m),0}}(m) \). The condition of reaching the node \( m \) in the fault-free path during fault simulation is the absence of all the faults in \( D_{P_{X(m),0}}(m) \). Denote by \( D_{C_{S}}(m) \) the set of faults consistent to the current faulty path from the initial node \( m_0 \) up to the node \( m \). For the nodes \( m \) on the fault-free path we have \( D_{C_{S}}(m) = D - D_{P_{X(m),0}}(m) \).

Let us denote by \( L \) the list of all nodes of the HLDD to be fault simulated. All the nodes met on the fault-free path are included into dynamic list \( L \). For carrying out fault simulation of the nodes in \( L \), either Procedure 2 (for terminal nodes) or Procedure 3 (for nonterminals) will be used. As the result of the procedures the list \( L \) will be updated. Fault simulation is terminated when the list \( L \) gets empty.

Procedure 2. Fault simulation of a terminal node \( m^{0} \in L \) with the function \( x = f(X(m)^0) = f(x_1, x_2, ..., x_p) \) for the set of complex input patterns \( T = (T_1, ..., T_p) \), \( T_j = \{P_{x_1,0}, (P_{x_1,1},D_{x_1}^{j}(m)), ..., (P_{x_p,1},D_{x_p}^{j}(m)), \} \), \( j = 1,2, ..., p \), where \( \forall j: D_{x_j} = (D_{x_j} - D_{P_{x_j}}(m^{0})) \cap D_{C_{S}}(m) \).

Procedure 3. Fault simulation of a nonterminal node \( m \in L \) with the variable \( x \) for the complex pattern \( T_{X(m)} = \{P_{X(m),0}, (P_{X(m),1},D_{x_1}^{j}(m)), ..., (P_{X(m),k},D_{x_k}^{j}(m)), \} \), \( \forall j: D_{x_j} = (D_{x_j} - D_{P_{x_j}}(m^{0})) \cap D_{C_{S}}(m) \), consists of the following:

- If \( m \) belongs to the fault-free path, and if \( D'_{X(m),1} \cup ... \cup D'_{X(m),k} \neq \emptyset \) then no nodes will be included into \( L \);
- If \( m \) does not belong to the fault-free path, and if \( D'_{X(m),1} \cup ... \cup D'_{X(m),k} \neq \emptyset \) then no nodes will be included into \( L \); for the new node \( m' \) in \( L \) we calculate: \( D_{C_{S}}(m') = D_{C_{S}}(m) \cup D_{x_1}, \) and \( D_{C_{S}}(m') = D_{C_{S}}(m) \).
- If \( D'_{X(m),i} \neq \emptyset \), all the nodes \( m' \), where \( e = P_{X(m),i} \), i.e., \( D'_{X(m),i} \neq \emptyset \), will be included into \( L \); for all these nodes we calculate \( D_{C_{S}}(m') = D_{C_{S}}(m) \cup D'_{X(m),i} \).

As a result of the fault simulation by Procedures 2 and 3 we create a complex pattern for the variable \( x : T_i = \{P_{x,i}, D_{x,i} \} \)
the faults propagated to TC, a subset of faults: fault free paths are shown by bold lines. The edges on paths in Tq are complex patterns: TA

Example: Consider the HLDD Gₐ in Fig.3 with a set of complex patterns:

Figure 4. Observability analysis on the HLDD

4 Experimental results

Comparative experiments between the HLDD-based code coverage analysis tool implemented in this paper and a popular HDL commercial simulation tool were carried out. Table 1 presents the circuits from the ITC99 benchmark family [12] and the Greatest Common Divisor (GCD) example to run the experiments.

Table 2 shows the comparison between traditional code coverage assessment (statement, branch and toggle coverage) carried out by a state-of-the-art commercial HDL simulator and by the HLDD-based simulator (implementing node, edge, toggle coverage, respectively). The code has been exercised using random set of stimuli of different length. The experiments were run on AMD Athlon 64 Processor 3000+, 1.80 GHz, 2.00GB of RAM, Windows XP.

While there is no definite advantage of the speed of basic logic simulation of benchmarks to either of the tools it should be noted that the overhead of coverage checking in Modelsim is much higher than in the case of HLDDs (See columns (5) and (8)). When HLDDs have coverage calculation overhead for 10000 patterns in a 1 to 4 % range, the commercial simulator uses 28 up to 78 % extra time. This result proves HLDD based code coverage analysis especially promising. As a future work we plan to incorporate the advanced features from event-driven cycle-based HLDD simulation algorithms [5] into our coverage analysis tool.

5 Conclusions

The paper presented a new approach to analyzing validation code coverage metrics using High-Level Decision Diagrams. A technique was proposed, where fast HLDD based simulation was extended to support code coverage analysis. We showed how classical code coverage metrics can be mapped to HLDD constructs. Moreover, we introduced an approach for observability coverage analysis on HLDD models. Experiments on ITC99 benchmark circuits indicated the feasibility of the proposed approach.

Acknowledgements

The work has been supported partly by EC FP 6 research project VERTIGO FP6-2005-IST-5-033709, Enterprise Estonia funded ELIKO Development Center and Estonian SF grants 6717, 7068 and 7483.
Table 1. ITC99 benchmark circuits

<table>
<thead>
<tr>
<th>design</th>
<th># of lines</th>
<th># of inputs</th>
<th># of outputs</th>
<th># of signals</th>
<th># of HLDD nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>b00</td>
<td>76</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>37</td>
</tr>
<tr>
<td>b04</td>
<td>84</td>
<td>6</td>
<td>1</td>
<td>14</td>
<td>58</td>
</tr>
<tr>
<td>b09</td>
<td>102</td>
<td>4</td>
<td>1</td>
<td>9</td>
<td>44</td>
</tr>
<tr>
<td>b10</td>
<td>169</td>
<td>10</td>
<td>3</td>
<td>14</td>
<td>116</td>
</tr>
</tbody>
</table>

Table 2. Comparison of traditional and HLDD-based code coverage measurement execution times

| design | test length (2) | Commercial HDL simulator | | | | HLDD simulator | | |
|--------|-----------------|---------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|
|        |                 | w/o coverage (3)       | w coverage (4)  | ratio (4)/(3)   | w/o coverage (6) | w coverage (7)  | ratio (7)/(6)   |
|        |                 | simulation time, s       | simulation time, s |       |                |                |       |
| b00    | 1000            | 0.0053                   | 0.0061           | 1.151           | 0.016           | 0.020           | 1.25            |
|        | 5000            | 0.0137                   | 0.0173           | 1.263           | 0.046           | 0.048           | 1.043           |
|        | 10000           | 0.0243                   | 0.0311           | 1.280           | 0.099           | 0.100           | 1.010           |
| b04    | 1000            | 0.0051                   | 0.0060           | 1.176           | 0.019           | 0.022           | 1.158           |
|        | 5000            | 0.0131                   | 0.0166           | 1.267           | 0.051           | 0.053           | 1.039           |
|        | 10000           | 0.0227                   | 0.0300           | 1.322           | 0.106           | 0.107           | 1.009           |
| b09    | 1000            | 0.0056                   | 0.0077           | 1.375           | N/A*            | 0.007           | N/A*            |
|        | 5000            | 0.0151                   | 0.0262           | 1.735           | 0.010           | 0.012           | 1.200           |
|        | 10000           | 0.0270                   | 0.0483           | 1.789           | 0.023           | 0.024           | 1.043           |
| GCD    | 1000            | 0.0052                   | 0.0061           | 1.173           | N/A*            | 0.003           | N/A*            |
|        | 5000            | 0.0135                   | 0.0178           | 1.319           | 0.015           | 0.016           | 1.067           |
|        | 10000           | 0.0240                   | 0.0316           | 1.317           | 0.031           | 0.032           | 1.032           |

* - N/A : the run time was too short to measure.

References