An Observability Branch Coverage Metric Based on Dynamic Factored Use-Define Chains

Tao Lv Ling-yi Liu Yang Zhao Hua-wei Li Xiao-wei Li
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
{lvtao, lingyiliu, zhaoyang, lihuawei, lxw}@ict.ac.cn

Abstract

In this paper we propose an observability branch coverage metric (OBCM) based on dynamic factored use-define chains, along with its evaluation method. This technique exploits the efficiency of data-flow analysis rather than methods like fault simulation. Hence it can be easily integrated into HDL compilers or simulators. Experimental results show that OBCM can provide more meaningful coverage data for functional verification than traditional branch coverage metric (BCM).

1. Introduction

Up to now, verification has become the dominant cost in the design process. Progress in formal techniques has partially alleviated the problem, but dynamic simulation continues to be the primary means of functional verification. However, it is usually infeasible to simulate all possible inputs for modern designs. Therefore, there is a well-defined need for coverage metrics: firstly, coverage quantitatively measures the “goodness” of simulation vectors; secondly, it can figure out what parts of the design have not been verified adequately, which will help to generate additional vectors.

Behavior coverage metrics are mature and widely utilized in commercial tools. Code coverage metrics, such as statement and branch coverage metrics, are common in functional simulation. They aim to evaluate whether the concerned code structures are exercised or not by given test patterns. Another category of behavior coverage is based on finite state machine (FSM) structure. The objects they evaluate are states, transitions, or transition paths of FSMs [2]. There are also metrics based on circuit structures, such as toggles of logics [2], register-to-register paths [1]. Finally, there are functional coverage metrics aiming at structures about function, e.g. interesting execution scenarios, and functional points in test plan.

Bug coverage is an important measure of verification, but it is currently poorly understood. Examples of fault models include gate (or input) omission, insertion, substitution and so on, which are derived from the similarity between hardware design verification and physical fault testing [4]. Creating a meaningful bug model requires a deep understanding of the design errors that arise in practice, which is a great challenge.

Observability coverage is a kind of behavior coverage. It concentrates on not only whether some structures are exercised, but also whether their effect is propagated to outputs. In functional simulation, it is often impractical to watch the waveforms or behaviors of all internal variables, which is a huge undertaking for modern designs. In automated design verification, the simulation values of key variables, which usually are the output variables of modules, are compared with expected values to discover design errors. Therefore only bugs whose effect could propagate to module outputs can be detected. Not taking observability into account can result in an artificially high reading of coverage and a false sense of security.

Unfortunately, nowadays coverage metrics are not very satisfactory. The observability assessment algorithms proposed as yet, focus mostly on observability-based statement coverage metric, rarely addressing the observability for branch coverage. It is well known that achieving complete branch coverage is a minimum requirement for simulation. In this paper, we propose an observability-based branch coverage metric and its evaluation method, taking the advantage of dynamic factored use-define chains (DFUD chains) [7][12].

The remainder of this paper is organized as follows. Section 2 introduces related work. Section 3 illustrates
the needed background from data flow analysis. Section 4 describes the observability model based on DFUD chains. Section 5 develops the observability evaluation method. Experimental results are presented in Section 6 and conclusion in Section 7.

2. Related work

2.1 Observability coverage metrics

Devadas et al. [5] and Fallah et al. [6] address observability information by tags. That approach is akin to fault simulation in circuit testing: a single tag is injected in any particular location in assignment statements, and propagated to the output. Tags are something additional to variables, so they must be handled specially. A dedicated simulation calculus is used in [5], and a two-phase approach in [6]. These methods are based on a mathematical structure for tags, in which both positive and negative tags are considered. It will be too time consuming if the magnitude of tags is taken into consideration. An unknown tag, represented by “?”, is used to simplify such situation.

Jiang et al. [9] develop a probabilistic observability measure to statement coverage metric. This method provides lower bound observability estimation. If the observability measure is high enough and achieves the threshold preset by users, the exercised statement is regarded as “covered”.

The paper [7] extends FUD chains from sequential software to concurrent hardware design, proposing EFUD chains, i.e. DFUD chains. Based on it, an observability statement coverage metric is presented.

All the work above focus on observability-based statement coverage evaluation, saying nothing about the observability for branches. In this paper, we propose an evaluation method for the observability of branches based on DFUD chains.

2.2 Software testing

As early as about 20 years ago, data flow techniques were utilized in software testing. It was used to detect data flow anomalies, for example, to see whether a variable is used without first being defined. There were also some coverage metrics in that field, e.g., covering all DU chains.

Ntafos [3] presents a method that uses data flow analysis to generate the required elements. Required element testing is a class of testing strategies that allow the various approaches to testing, e.g. structural testing, black-box testing, etc., to be combined. It consists of a structural component and a functional component. Ntafos defines the k-dr interaction and takes it as the basic structural unit in required element testing. A k-dr interaction consists of k-1 variables X1, X2, ..., Xk-1, and k distinct statements s1, s2, ..., sk, such that there is a path that visits the statements in the given order, and statement sk utilizes Xki to define Xi, for 1 < i ≤ k.

The k-dr interaction can be regarded as an extension of DU chains. In this paper, we utilize a similar structure, DFUD chains. However, there are many differences between them. Firstly, Ntafos uses k-dr interactions as a general structure representation to generate required elements. Thus there is no definite meaning of k-dr interactions and the position of the last reference in k-dr interactions is variable. In our method, DFUD chains are associated with observability at register transfer level (RTL), and the last element in the chain is a definition of an output variable, but not a reference as is in k-dr interactions. Secondly, the value of k is a constant for corresponding required element strategy. But in our work, the lengths of the lists vary with the distances between internal variables and outputs. Finally, our testing objects are Hardware Description Language (HDL) codes, but not software programs. Accordingly, we modify traditional data flow techniques to meet the characteristics of HDL codes, which will be introduced in section 5.

3. Dynamic factored use-define chains

First of all, we introduce some related terms in data flow analysis techniques. Data flow analysis is a static analysis of the program that provides information about actions taken on the program variables. The actions that are of interest in coverage metric here are definitions (D) and uses (U) of variables. A definition of a variable \( x \) is a statement that assigns a value to \( x \). A variable \( x \) is used (referenced) in a statement, if the statement requires the value of \( x \) to execute. For example, in the statement \( \text{outp} <= \text{a} & \text{b} \), \( \text{outp} \) is defined, and \( \text{a} \) and \( \text{b} \) are used. We say a definition \( d \) in statement \( s_1 \) reaches another statement \( s_2 \), if it is not re-defined along the path from \( s_1 \) to \( s_2 \). It is often convenient to store the reaching definition information as “use-definition chains” (UD chains), which are lists, for each use of a variable, of all the definitions that reach that use. FUD chains are an improved form of UD chains. It has two important properties. The first is that each use of a variable is reached by a single definition. The second is that special merge operators called \( \phi \) -terms are inserted into control-flow merge points when there exist multiple reaching definitions.

Fig. 1 (a) is a segment of sequential codes usual in software program. In Fig. 1 (b), each variable
Definition and use is labeled by a subscript. Fig. 1 (c) shows UD chains with the links from each use to all reaching definitions. For example, \( K_6 \rightarrow 5,1 \) means that \( K_6 \) may use the value of \( K \) defined as \( K_5 \) or \( K_7 \). Finally, Fig. 1 (d) explains FUD chains. The *endif* is a control-flow merge point, therefore a \( \phi \)-term, \( \phi(K)_8 \rightarrow 5,1 \), is added to keep two distinct reaching definitions for \( K_8 \), and thus \( K_8 \) in Fig. 1 (d) has only one reaching definition, i.e., \( \phi(K)_8 \).

1. \( K = 0 \) \quad 1. \( K_1 = 0 \)
2. if \( T > K \) then \quad 2. if \( T_2 > K_3 \) then
3. \( K = K + 1 \) \quad 3. \( K_5 = K_4 + 1 \)
4. endif \quad 4. endif
5. \( M = K + 2 \) \quad 5. \( M_2 = K_8 + 2 \)

(a) Original code \quad (b) With reference labeled

1. \( K_1 = 0 \) \quad 1. \( K_1 = 0 \)
2. if \( T_2 > K_3 \rightarrow 1 \) then \quad 2. if \( T_2 > K_3 \rightarrow 1 \) then
3. \( K_7 = K_4 \rightarrow 1 + 1 \) \quad 3. \( K_5 = K_4 \rightarrow 1 + 1 \)
4. endif \quad 4. endif
5. \( M_3 = K_9 \rightarrow 5,1 + 2 \) \quad 5. \( M_3 = K_9 \rightarrow 5,1 + 2 \)

(c) UD chains \quad (d) FUD chains

Figure 1. Example of UD chains and FUD

DFUD chains still preserve the two original properties of FUD chains, and they have a new property: for each \( \phi \)-term, its DFUD chain can point out at some cycle which reaching definition is active.

**Definition 1.** The set of DFUD chains for a module can be represented as a structure \( <S, F> \), where \( S \) is the set of original FUD chains, and \( F: \langle \phi, c \rangle \rightarrow I \) is the function mapping each \( \phi \)-term in \( S \) to an integer \( I \) which corresponds to the active branch at cycle \( c \) [12].

For example in Fig. 1 (d), \( \phi(K)_8 \rightarrow 5,1 \), tells us that there are two possible reaching definitions for \( \phi(K)_8 \), namely \( K_5 \) and \( K_7 \). If \( T = 1 \), the “if” branch will be taken, and the DFUD chain for \( \phi(K)_8 \) will record that at this time \( K_5 \) is the active definition. More details about DFUD chains can be found in [12].

4. **Observability branch coverage metric**

4.1 The assumption

In current hardware design flow, RTL is usually a necessary phase. Therefore, our work focuses on RTL designs. Our basic premise is that it is of high possibility for bugs to transfer through data paths in RTL designs. Hence if a variable is referenced in an assignment statement, we think that its bug, if there is any, could be propagated to the variable in the left-hand with a great probability. Fortunately, it is true in many cases. For example, in many RTL designs the arithmetic operators “+” and “−” are rarely utilized, and operations are often taken as “&”, “|”, “&&”, etc. of arrays of bits. Hence, blocking one bit is usually of little impact on bugs’ propagation.

4.2 Observability of variables

**Definition 2.** A variable \( x \) data depends on a variable \( y \) if an assignment statement \( s \) uses the value of \( y \) to compute and assign a value to \( x \).

Apparently, the left-hand variable in an assignment statement data depends on its right-hand variables.

**Definition 3.** A variable \( x \) in a statement \( s \) is control dependent on a variable \( c \) if the value of \( c \) determines whether \( s \) is executed or not.

For example in Fig. 1 (b), \( K_5 \) in line 3 is control dependent on \( T_2 \) and \( K_3 \).

**Definition 4.** The observable point set of a segment of codes \( C \), denoted as \( OPS(C) \), is defined as follows: [12]

1) A variable \( v \) is a direct observation point (DOP) if through it simulation results are checked. The set of DOPs is denoted as \( DOPS(C) \).

\[ v \in DOPS(C) \subseteq OPS(C) \]

2) An internal variable \( w \) is observable if a known observable variable data or control depends on it.

\[ \left\{ w \mid \exists v \in OPS(C) \left( v \text{ data control depends on } w \right) \right\} \subseteq OPS(C) \]

3) An internal variable \( w \) is observable if it can be arrived through the DFUD chain from a known observable variable.

\[ \left\{ w \mid \exists v \in OPS(C) \left( v, w \in DFUDS \right) \right\} \subseteq OPS(C) \]

Where, \( DFUDS \) is the DFUD chain set of \( C \), \( \langle v, w \rangle \in DFUDS \) means that there is a DFUD chain pointing from \( v \) to \( w \). For example, in Fig. 1 (d), \( K_4 \rightarrow 1 \) can be denoted as \( \langle K_4, K_7 \rangle \).

Assume that \( v_b \) is a DOP, and there are DFUD chains \( \langle v_b, v_1 \rangle, \langle v_1, w \rangle \), and statement \( v_1 = v_i + 1 \) uses \( v_1 \) to define \( v_i \). Firstly, \( v_i \) is observable according to definition 4.3). Then \( v_i \) is observable since \( v_i \) data depends on it. Finally \( w \) is observable because \( \langle v_1, w \rangle \) is a DFUD chain and \( v_1 \) is observable. It is to say, the value of \( w \) can be propagated to \( v_1 \), then \( v_i \) has an effect on \( v_1 \), finally definition \( v_i \) reaches the use \( v_b \).

Along this path, variable \( w \) is observable through \( v_b \).

4.3 Observability branch coverage metric

**Definition 5.** For a general branch, the variables on its execution path are its related variables; for an empty branch, the variables on execution paths of its brother-branches, i.e. the branches that divericate and converge at the same points with the null branch, are its related variables.
Take the code segment in Fig. 1 (a) as an example. When T is larger than K, the “if” branch is taken, and variable M will utilize K’s value which is computed in line 3. When T is less than K, the “else” branch is taken, i.e. no new value is assigned to K, and it holds the original value 0. The effect of different branch is shown via the behavior of K. Therefore K is the related variable for both the two branches. In other words, K is assigned a new value in the “if” branch, so it is the related variable of the “if” branch; as for the corresponding empty “else” branch, its effect is that it takes no action on K but its brother-branch does, and accordingly K is its related variable, too.

Definition 6. A branch is observed if it is executed and one of its related variables has been observed. Branch structures reflect designers’ consideration about the different actions taken on the related variables in different conditions. Therefore, if one related variable of a branch is observed, we could say the effect of the branch is observed through the distinct behavior of the related variable.

Definition 7. For a code segment C, its observability-based branch coverage (OBC) is defined as following:

\[
OBC(C) = \frac{\text{the number of observed branches}}{\text{total number of branches}} \times 100\% 
\]

If the OBC is less than 100%, it means that either some branch is not activated in simulation, or it is exercised but its effect is blocked and not propagated to DOPs (module outputs). In other words, its effect is not checked.

5. Coverage evaluation

There are ready algorithms for constructing FUD chains for software programs, for instance ones in [8] (p.175 and p.178). They are built in the same way as that in compilers.

We introduce some mechanisms to deal with special characteristics of HDL codes.

Firstly, an HDL program is a non-halting reactive system. A START node and an END node are added for each process in a design. The START node locates before the node corresponding to the first statement in the process; The END node locates after the node corresponding to the last statement. Furthermore, a control flow edge is added pointing from the END node to the START node, which represents the non-halting characteristic of “always” blocks. When analyzing DFUD chains, it is known that there may be some definitions propagating from the end of a process to the entry of it.

Secondly, processes in HDL codes communicate with each other through signals shared between them. Considering the well-known coding style for synthesis, we take the assumption that a variable is defined in only one process. For each process, we add pseudo-references in its END node for each variable defined in it, thus all variable definitions will reach the END node via traditional FUD chains construction, and DFUD chains will record the actual values in different cycles. Therefore, inter-process analysis is implemented by pointing the DFUD chain from a variable reference in one process to the END node of the process that defines the variable. As for intra-process analysis, it can be implemented with techniques in traditional software analysis, since procedural statements in a process are executed sequentially.

Our algorithm could be implemented as a plug-in of the simulator, and we take event-driven strategy to evaluate observability-based branch coverage.

The event-driven analysis strategy includes: In the event that control flow of the program runs to a merge node, update its DFUD chains to trace control flow; In the event that the value of a DOP is updated, analyze observability coverage along with DFUD chains starting from the DOP.

Other concurrent statements such as continuous assignments are treated as single-line unique processes that are sensitive to their respective right-hand signals.

The Verilog code in Fig. 2 (a) was taken as an example to demonstrate our algorithm. Fig. 2 (b) shows its CFG and DFUD chains. Circular nodes are referred as computation nodes, each of which corresponds to a Verilog statement. Node 3’ is the merge node for node 3, corresponding to an implicit endif statement. Solid edges represent control flow, and dash edges represent DFUD chains.

\( \text{rdy} \) is an input of the model, and its value is 0 for the given test vector. \( \text{outp} \) is an output variable. Consequently it is a direct observation point. The process of OBC evaluation is as follows.

1. Since \( \text{rdy} = 0 \), line 6 of the Verilog code is exercised, which triggers the event to update the DFUD chain of \( \phi_s \) and thus records that the “else” branch is executed in current cycle. It means that in Fig. 2 (b), the right chain of \( \phi_s \) in node 3’ (i.e. the dash line from node 3’ to node 6) is the latest definition for s.
2. When line 7 is simulated, it triggers the event to analyze observability coverage since \( \text{outp} \) is a DOP.
   (1) The definition of \( \text{outp} \) is denoted as observed, since it is DOP and it is activated.
   (2) Considering definition 4.2, the use of \( s \) in line 7 is observed. We trace DFUD chain of \( s \) backward, i.e. the dash edge from node 7 to \( \phi_s \) in node 3’, and reach node 3’. The DFUD chain in \( \phi_s \) indicates that the right
chain is valid, so we trace further to node 6. Therefore
the definition of \( s \) in line 6 is denoted as observed. And
then the “else” branch of conditional statement 3 is
observed, since \( s \) is its related variable.

1 always @(posedge clock)
2 begin
3 if (rdy)
4    \( s = 0 \);
5 else
6    \( s = 1 \);
7    outp = s;
8 end

(a) Verilog code

Figure 2. Example for OBC evaluation

(3). In line 6, \( s \) is assigned a constant value, i.e. its
DFUD chain is null. Its data dependency analysis
finishes.

(4). It is because \( \text{rdy}=0 \) that statement 6 is executed,
but not statement 4. Hence the variable \( s \) is control
dependent on variable \( \text{rdy} \) in line 3. Hence, the use of
\( \text{rdy} \) in line 3 is observed.

(5). Variable \( \text{rdy} \) is an input variable, which means
its DFUD chain is null. Current evaluation finishes.
3. After simulation, we compute the observability-
based branch coverage. It is clear that one of the two
branches is observed. Hence, the OBC is 50%.

Note that in step 2(2) above, the value of \( s \) in
CURRENT cycle is traced because of the blocking
assignment to \( \text{outp} \). If it is a non-blocking assignment,
the algorithm will first direct to the END node, and
then trace along the execution path in previous cycle.

6. Experiments and discussion

A prototype system called OCM_Statistics has
been developed. It interacts with the simulator via
Verilog PLI, and integrates a front-end tool, i.e. VRM,
to parse Verilog codes and to implement intra-process
analysis. OCM_Statistics can evaluate observability-
base branch coverage for Verilog codes, including
one-process designs and multi-process designs.

6.1 Metric comparison

b03 is an arbiter, which is modeled as an FSM by a
“case” statement. Requests are analyzed and arbitrated
in one state, and then the decision is propagated to
output in other states. Therefore, the effect of the
analysis state cannot be observed unless its internal
arbitration decision has propagated to outputs, which
will take several cycles. As a result, its observability-
based branch coverage (OBC) for several random
vectors is not so optimistic as traditional branch
coverage (BC) for the same vector set, as is illustrated
in Fig. 3.

(b) CFG and DFUD chains

Figure 2. Example for OBC evaluation

ITC-99 benchmark [10] circuits are translated into
Verilog version and have been taken as our
experimental objects. They are RTL descriptions, and
are described briefly in Table 1. The outputs of each
model are taken as its DOPs. And different metrics, i.e.
BCM and OBCM, have been used to evaluate the same
100 random vectors for a benchmark circuit, as in
Table 2.

Table 1. ITC-99 benchmark description

<table>
<thead>
<tr>
<th>Circuits</th>
<th>#Lines</th>
<th>#Processes</th>
<th>#Gates</th>
<th>#FF description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b01</td>
<td>84</td>
<td>1</td>
<td>45</td>
<td>FSM</td>
</tr>
<tr>
<td>b03</td>
<td>122</td>
<td>1</td>
<td>150</td>
<td>Arbiter</td>
</tr>
<tr>
<td>b04</td>
<td>110</td>
<td>1</td>
<td>480</td>
<td>compute min-max</td>
</tr>
<tr>
<td>b05</td>
<td>276</td>
<td>3</td>
<td>608</td>
<td>elaborate ram</td>
</tr>
<tr>
<td>b06</td>
<td>140</td>
<td>1</td>
<td>66</td>
<td>interrupt handler</td>
</tr>
<tr>
<td>b09</td>
<td>104</td>
<td>1</td>
<td>131</td>
<td>serial-serial converter</td>
</tr>
<tr>
<td>b11</td>
<td>115</td>
<td>1</td>
<td>366</td>
<td>scramble string</td>
</tr>
<tr>
<td>b12</td>
<td>549</td>
<td>4</td>
<td>1000</td>
<td>simon game</td>
</tr>
<tr>
<td>b13</td>
<td>333</td>
<td>5</td>
<td>309</td>
<td>interface to meteo sensor</td>
</tr>
<tr>
<td>b15</td>
<td>752</td>
<td>3</td>
<td>6931</td>
<td>subset of 80386</td>
</tr>
</tbody>
</table>

The results in Table 2 serve to illustrate that
because of additional observability information, OBC
is not so optimistic excessively as traditional BC does.
The typical reason is that some combinational logics
are activated in simulation, but their values are not
propagated to registers because of control logics.

Table 2. Comparison of coverage metrics

<table>
<thead>
<tr>
<th>Circuits</th>
<th>#OdPs</th>
<th>BC(%)</th>
<th>OBC(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b01</td>
<td>2</td>
<td>100.0</td>
<td>96.4</td>
</tr>
<tr>
<td>b03</td>
<td>1</td>
<td>100.0</td>
<td>92.9</td>
</tr>
<tr>
<td>b04</td>
<td>1</td>
<td>100.0</td>
<td>92.9</td>
</tr>
<tr>
<td>b05</td>
<td>1</td>
<td>100.0</td>
<td>92.9</td>
</tr>
<tr>
<td>b06</td>
<td>2</td>
<td>86.6</td>
<td>69.7</td>
</tr>
<tr>
<td>b09</td>
<td>3</td>
<td>73.7</td>
<td>73.7</td>
</tr>
<tr>
<td>b11</td>
<td>4</td>
<td>73.7</td>
<td>73.7</td>
</tr>
<tr>
<td>b12</td>
<td>5</td>
<td>73.7</td>
<td>73.7</td>
</tr>
<tr>
<td>b13</td>
<td>6</td>
<td>73.7</td>
<td>73.7</td>
</tr>
<tr>
<td>b15</td>
<td>7</td>
<td>73.7</td>
<td>73.7</td>
</tr>
</tbody>
</table>

Fig. 4 demonstrates the difference between BC
and OBC further. The left is code segment, and the
right is its CFG.
For the code segment shown in Fig.4, we assume that \( d \) is output variable, i.e. the DOP, and \( x, y \) are both positive in the first vector, and both negative in the second one. Thus the two left branches in its CFG are executed firstly, and following is the two right branches. As a result, all the branches are activated, and BC is 100%. However, in the second vector, the value of \( b \) is not propagated to DOP \( d \). Hence OBC is not so misleading, just 75%, i.e. the right branch of the first “if” statement is not observed.

![Example for metric comparison](image)

Figure 4. Example for metric comparison

For the code segment shown in Fig.4, we assume that \( d \) is output variable, i.e. the DOP, and \( x, y \) are both positive in the first vector, and both negative in the second one. Thus the two left branches in its CFG are executed firstly, and following is the two right branches. As a result, all the branches are activated, and BC is 100%. However, in the second vector, the value of \( b \) is not propagated to DOP \( d \). Hence OBC is not so misleading, just 75%, i.e. the right branch of the first “if” statement is not observed.

### 6.2 Performance

**Table 3. Time overhead of OBC evaluation**

<table>
<thead>
<tr>
<th>Circuits</th>
<th>b01</th>
<th>b03</th>
<th>b04</th>
<th>b05</th>
<th>b06</th>
<th>b09</th>
<th>b11</th>
<th>b12</th>
<th>b13</th>
<th>b15</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1(s)</td>
<td>9</td>
<td>10</td>
<td>20</td>
<td>22</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>16</td>
<td>17</td>
<td>25</td>
</tr>
<tr>
<td>T2(s)</td>
<td>28</td>
<td>34</td>
<td>110</td>
<td>69</td>
<td>62</td>
<td>44</td>
<td>17</td>
<td>117</td>
<td>78</td>
<td>35</td>
</tr>
<tr>
<td>overhead</td>
<td>2.11</td>
<td>2.40</td>
<td>4.50</td>
<td>2.14</td>
<td>5.20</td>
<td>3.40</td>
<td>0.89</td>
<td>6.32</td>
<td>3.59</td>
<td>0.40</td>
</tr>
</tbody>
</table>

Table 3 shows the overhead of computing OBC using OCM_Statistics. An open-source simulator, Icarus Verilog Compilation System [11], is used in this experiment. The time required to simulate the original HDL model for 1,000,000 vectors (T1) is given in the second line. The same vectors are used to compute OBC, and the time (T2) is shown in the third line. All times correspond to seconds on a PC with 512 MB of RAM running at 3.0 GHz. The time overhead is computed as following.

\[
\text{Time overhead} = \frac{\text{Coverage evaluation time} - \text{original simulation time}}{\text{original simulation time}}
\]

In this experiment, our prototype works as a plug-in of the simulator. Since our algorithm makes use of many compiling techniques, it can be integrated into HDL compilers or simulators directly, which will reduce the overhead to a greater extent.

### 7. Conclusions and future work

In this paper we utilize DFUD chains to model observability, and propose an observability-based branch coverage metric, along with its evaluation algorithm. Although it is optimistic in some cases, e.g. “&” of one-bit signals, experimental results show that OBCM can provide more meaningful coverage data for RTL functional verification than traditional BCM. Moreover, this technique exploits the efficiency of data-flow analysis. Therefore it can be easily integrated into HDL compilers or simulators to improve its computing efficiency.

Future work would focus on generating vectors with high coverage based on OBCM, and improving the computing efficiency of our OBCM evaluation algorithm.

**REFERENCES**


