Overview

1. Introduction
2. Motivation
3. Key Contributions
4. Methodology
5. Experimental Results
6. Conclusion
Assertion-Based Verification (ABV) plays a significant role in the design validation flow.

Some of the popular formal languages for assertion specification include Forspec, PSL [1] and SVA [2].

Assertion specifications are used to verify given implementations through:
- formal property verification (FPV) techniques like model checking
- dynamic assertion-based verification (ABV)

Linear Temporal Logic (LTL), is the foundation of most of the assertion specification languages used today.
Motivation

- In a traditional simulation based ABV evaluation paradigm
  - For each assertion, as per LTL semantics, an automaton is created
  - At each cycle, each assertion is evaluated separately by evaluation engine by reading the values of propositional variables
  - Inefficient, time consuming, memory overhead
Key Contributions

**Main Contribution**

We propose a substantially different mechanism for assertion evaluation over simulation runs

- We attempt to exploit the overlap in signal variables across the set of assertions
- We leverage the LTL semantics to infer results across time
Methodology

- We propose a novel methodology for assertion simulation with two phases
- In first phase, called preprocessing phase:
  - Preprocess the assertion set
  - Construct shared graph structure
  - Store the requisite information in a lookup table
- In second phase, called assertion evaluation phase:
  - Using lookup tables, we evaluate the assertions
- We assume the assertions are in Negation Normal Form (NNF)
Some details about shared graph structure

- Shared graph is a directed acyclic graph
- Consists of three kind of nodes:
  - **Input Node**: Each propositional variable, present in the assertions, is assigned an input node.
    - In-degree of an input node is 0
  - **Internal Node**: Associated with a subexpression. consists of an operator
    - Non-zero in and out degree
  - **Assertion Node**: Corresponds to a particular assertion. Holds the results of assertions across cycles.
    - Out-degree is 0
Some details about shared graph structure

- Edges are annotated with $val_{in}/val_{out}$;  
  $val_{in}/val_{out} \in \{True(T), False(F), Unknown(U)\}$

- Source node has value $val_{in}$; $val_{out}$ is propagated to destination node

- Two types of edges:
  - **Strong Edge**: Causes a destination node to be assigned a fixed value,  
    does not change in future clock cycles.

  - **Ordinary Edge**: Value of destination node may change across cycles

- The level of a node is defined as:
  - Level of input node $= 0$
  - $= \text{MAX}(\text{level of immediate predecessor}) + 1$
Basic Building Block: Rule for AND operator

- If one operand is false, the result is False, irrespective of the values of other operands
  - Edges are marked by $F/F$
- If source is unknown (may happen for temporal expr.)
  - Edges are marked by $U/U$
- Default value of the destination is $T$
- If $F$ is propagated by an edge, $F$ is assigned to the internal node (corresponding to AND operator)
- If the node does not hold $F$ and $U$ is propagated by an edge, $U$ is assigned to the node
- Otherwise the node holds default value $T$, implies no operands are $F/U$
Signifies $\phi_1$ has to hold $T$ at least until $\phi_2$

- The default value of node $\mathcal{U}$ is $U$
- If internal $\lor$ node is $F$, $\mathcal{U}$ operator node is set to $F$
- If the node $\mathcal{U}$ holds $U$ and $\phi_2$ is $T$, UNTIL operator node will be True
Basic Building Block: Rule for NEXT operator

- We do not explicitly create a node for NEXT operator
- We associate the NEXT operator with the variable itself
- We introduce the notion of a delay corresponding to a node
  - The delay of an input node is the number of NEXT operators preceding the corresponding variable present in this node
  - The delay of an internal node is the maximum delay of its predecessor nodes
Look up table generation

- For each source node $n_i$
  - We store three lists: True list, False list and Unknown list
  - For an edge $(n_i, n_j)$ with edge marking $val_{in}$, $val_{out}$
    - We store node $n_j$ with $val_{out}$ in the associated $val_{in}$ list of $n_i$
  - We set the value of successor node of $n_i$ by looking at the $val_{in}$ list of $n_i$
- Initially the assertions are preprocessed and lookup table is generated
An Example

$P_1: G(a \lor c); P_2: G(a \land Xb); P_3: F(c \lor d \lor e)$

Figure: Shared Graph Data Structure

<table>
<thead>
<tr>
<th>Level</th>
<th>Type</th>
<th>Node</th>
<th>List type</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Input</td>
<td>a</td>
<td>True False Unknown</td>
<td>$n_1$ $n_3$ $n_1, n_3$</td>
</tr>
<tr>
<td>0</td>
<td>Input</td>
<td>b[1]</td>
<td>False</td>
<td>$n_3$</td>
</tr>
<tr>
<td>0</td>
<td>Input</td>
<td>c</td>
<td>True Unknown</td>
<td>$n_1, n_5$ $n_1, n_5$</td>
</tr>
<tr>
<td>0</td>
<td>Input</td>
<td>d</td>
<td>True Unknown</td>
<td>$n_5$ $n_5$</td>
</tr>
<tr>
<td>0</td>
<td>Input</td>
<td>e</td>
<td>True Unknown</td>
<td>$n_5$ $n_5$</td>
</tr>
<tr>
<td>1</td>
<td>$\lor$</td>
<td>$n_1$</td>
<td>False</td>
<td>$n_2[s]$</td>
</tr>
<tr>
<td>1</td>
<td>$\land$</td>
<td>$n_3[1]$</td>
<td>False</td>
<td>$n_4[s]$</td>
</tr>
<tr>
<td>1</td>
<td>$\lor$</td>
<td>$n_5$</td>
<td>True</td>
<td>$n_6[s]$</td>
</tr>
<tr>
<td>2</td>
<td>$G$</td>
<td>$n_2$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>$G$</td>
<td>$n_4[1]$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>$F$</td>
<td>$n_6$</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table: Look Up Table
In each cycle, a new thread is created for evaluation.

If the evaluation of an assertion starts at clock cycle $t_0$, each node $n_i$ at delay $t_d$ is evaluated at $(t_0 + t_d)^{th}$ cycle.

This method arranges nodes in order of the level to which it belongs.

The algorithm proceeds by initializing the nodes to their default values.

At clock cycle $t_i (\geq t_0)$, thread reads the simulation inputs for input nodes with temporal delay $(t_i - t_0)$.

Once all the nodes in the current level have been processed, the nodes in next level are taken up.
Implementation and Results

- We compare our proposed Level Based Simulator (LBS) against an in-house LTL simulator (LSim)
  - LSim implements assertions as individual monitors
  - Evaluates them with rudimentary simulation
  - Randomly generated simulation inputs, assertions and number of simulation cycles were used to compare the relative performances
  - The memory requirements were in the order of Kilo-bytes for both
  - The LUT-based evaluation mechanism made significant performance difference

<table>
<thead>
<tr>
<th>Exp</th>
<th>#Assertions</th>
<th>#Vars</th>
<th>#Sim. cycles</th>
<th>LBS</th>
<th>LSim</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>31</td>
<td>4</td>
<td>753</td>
<td>0.083</td>
<td>38.847</td>
</tr>
<tr>
<td>2</td>
<td>26</td>
<td>6</td>
<td>899</td>
<td>0.125</td>
<td>114.687</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>6</td>
<td>1159</td>
<td>0.154</td>
<td>148.99</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>18</td>
<td>1690</td>
<td>0.181</td>
<td>177.29</td>
</tr>
</tbody>
</table>

Table: Performance comparison of LBS vs LSim
LBS was evaluated on a subset of assertions from a commercial SVA-based assertion IP for the Open Cores Protocol (OCP) [3] with a rudimentary OCP design model, and assertions written in LTL. We tested the same using commercial ABV simulators like VCS [4] and Questa [5].

- Average runtime requirements were in the order of 10s
- Peak memory requirement was greater than ours
- For final run, both VCS and Questa ran out of memory

We do not furnish comparative results, since the high memory requirement may as well be attributed to the simulation data structure overhead. We could not find a direct way to get the peak memory required for assertion handling only.

<table>
<thead>
<tr>
<th>No of signals</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td>No of assertions</td>
<td>45</td>
</tr>
<tr>
<td>Look up Table Generation Time</td>
<td>0.455s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No of cycles simulated</th>
<th>Simulation time (in secs)</th>
<th>Peak memory (in mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.763</td>
<td>24.28</td>
</tr>
<tr>
<td>1000</td>
<td>0.2888</td>
<td>12.4906</td>
</tr>
<tr>
<td>10000</td>
<td>1.5862</td>
<td>87.4856</td>
</tr>
<tr>
<td>100000</td>
<td>11.7932</td>
<td>154.49</td>
</tr>
<tr>
<td>1000000</td>
<td>117.2896</td>
<td>169.992</td>
</tr>
</tbody>
</table>

Table: LBS performance for OCP assertions
We presented a new methodology for simulation of LTL assertions

The foundation of this work is based on inferring the evaluation results of the assertions

We are currently working on integrating our evaluation framework with commercial ABV simulators
References


Open Core Protocol. accellera.org/downloads/standards/ocp/.


Thank You!