EAST: Efficient Assertion Simulation Techniques

Debjyoti Bhattacharjee
Indian Statistical Institute Kolkata
Email: debjyoti08911@gmail.com

Soumi Chattopadhyay
Indian Statistical Institute Kolkata
Email: soumi_r@isical.ac.in

Ansuman Banerjee
Indian Statistical Institute Kolkata
Email: ansuman@isical.ac.in

***Abstract*— In the context of simulation-based verification, the Assertion-based Verification (ABV) methodology has become the technology of choice, with increasing proliferation of Verification / Assertion IPs for most commonly used protocols. To support the ABV flow, current generation simulators typically create threads for the assertions and evaluate each assertion separately by converting them into finite state automatons and monitoring their states during simulation. In this paper, we propose a different technique for assertion evaluation in a simulation-based verification flow. The proposed technique, EAST (Efficient Assertion Simulation Techniques), handles assertions in groups, instead of examining them in isolation, and achieves significant performance benefits. To this effect, our algorithm has a pre-processing phase (prior to simulation) which creates a shared data structure from the set of assertions using some simple rules, based on the assertion language operators. This is attached with the simulator and during simulation, at each evaluation cycle, EAST infers the decision of the assertions by a combination of lookup and substitution. We present our proposal using Linear Temporal Logic (LTL) assertions in this paper. Our prototype, EAST, achieves promising performance numbers in terms of both runtime and peak memory for both random and standard benchmark protocol designs.**

I. Introduction

In recent times, Assertion-Based Verification (ABV) is assuming a significant role in the design validation flow of chip design companies. Active participation from the design and EDA industries have led to the adoption of several formal languages for assertion specification. Some of the popular languages include Forspec [7], PSL [2] and SVA [4]. Assertion specifications written in these languages are used to verify given implementations, either through formal property verification (FPV) techniques like model checking or dynamic assertion-based verification (ABV) which is typically done by monitoring the properties over simulation runs. In this paper, we use ABV as the background for our work with assertions written using Linear Temporal Logic (LTL) [13], which is the foundation of most of the assertion specification languages used today in the verification community.

There is a rich body of literature [6], [8]–[10] for dynamic ABV of LTL and other LTL based languages. To the best of our knowledge, classical simulation-based verification engines [3] [5] typically translate each assertion to a finite state automaton, and thereafter deploy a thread based simulation strategy for dynamic assertion evaluation. In [9], PSL assertions are translated to Verilog and thread based monitoring is undertaken for simulation based assertion checking. In [8], assertions are compiled to deterministic automata and simulated in a thread-less uniprocessor environment. [14] proposes a methodology for temporal monitors for SystemC. Similar research has been proposed in [11], [12].

In this paper, we propose a substantially different mechanism for assertion evaluation over simulation runs. Our method, instead of treating assertions individually and creating monitors for them, groups assertions into clusters and replaces the run-time overhead significantly by a simple table lookup. Additionally, we provide bounds on the number of active threads to be created during simulation. Our methodology is motivated by the fact that a set of assertions for a design under test typically share multiple design variables between them, which leads to efficient clusters and a simple efficient methodology for substitution and inference-based assertion evaluation. This forms the backbone of our evaluation engine. Our methodology uses a shared graph data structure to represent these assertions considering the signals shared and their overlap across assertions over time. Our simulation methodology orders the evaluation of the nodes in this graph data structure in a way that we can infer the evaluation results of the assertions across multiple cycles by a simple table lookup and value substitution. In this paper, for simplicity of illustration and ease of demonstration, we demonstrate our methodology using the LTL operators. However, our method is generic enough to be extended to be seamlessly embedded into SVA / PSL simulation kernels as well.

The key contributions of this paper are as follows.

- A preprocessing methodology to generate a shared graph data structure from a set of LTL assertions
- An efficient thread based assertion evaluation strategy using the shared graph data structure
- Performance evaluation of the proposed methodology on random and industrial assertion test suites.

II. A Motivating Example

In logic, LTL is a temporal logic with modalities referring to time. LTL is built from a finite set of propositional variables, the logical operators {NOT($\neg$), AND($\wedge$), OR($\vee$)} and the temporal operators {NEXT(X), GLOBAL(G), EVENTUALLY($\mathcal{F}$), UNTIL($\mathcal{U}$) }). To motivate our case, we consider a simple example with 3 LTL assertions.

$[P_1] : G(a \vee c)$; $[P_2] : G(a \wedge Xb)$; $[P_3] : \mathcal{F}(c \vee d \vee e)$

In a traditional simulation based ABV evaluation paradigm, at each cycle, each assertion is evaluated separately by the assertion evaluation engine, by reading the values of the propositional variables and thereafter, monitoring the state

978-3-9815370-6-2/DATE16/

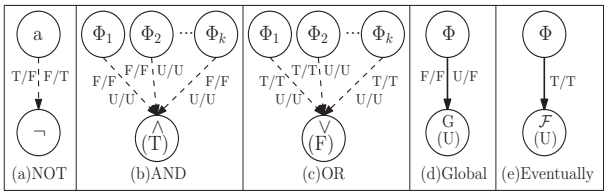

Fig. 1: Graph data structures for various operators

(accept / reject / not-yet-conclusive) of the corresponding automata created as per LTL semantics [13]. This is where our proposal comes into play, whereby, instead of evaluating assertions individually, we attempt to exploit the overlap in signal variables they share and leverage the LTL semantics to infer the results across time. For example, if c is found to be $True$ during evaluation of the assertion P_1, we can conclude the expression $(c \vee d \vee e)$ in assertion P_3 is also $True$. Similarly if a is found to be $False$ using evaluation of assertion P_1, we can immediately conclude the expression $(a \wedge Xb)$ in assertion P_2 will evaluate to $False$. Thus, we can see the presence of common variables across assertions can be used to minimize evaluation efforts of the assertion set and thereby speedup simulation.

In this paper, we present a novel methodology for assertion simulation consisting of two stages, namely a preprocessing stage and a simulation stage. The preprocessing stage processes the assertion set to store the requisite information in a look up table. This look up table can be visualized as a shared graph data structure. Using look up tables, instead of individual assertions for simulation in the simulation stage, we leverage the presence of common variables across assertions to accelerate the assertion evaluations. We begin by explaining the building blocks of the shared graph data structure and then present the preprocessing and simulation steps in Subsections III-A and III-B respectively. Thereafter, we present a couple of optimizations to speed up our methodology in Subsection III-C. In addition, we give a detailed walk through of our methodology on Example 1.

III. Detailed Methodology

Our methodology uses a look up table for simulation of the assertions which can be equivalently represented as a shared graph data structure. We assume that the assertions are present in the Negation Normal Form (NNF). Before we explain our methodology in detail, we present the building blocks of the graph, namely nodes, edges and basic rules of graph generation that will be used in the subsequent subsections.

Nodes : There are three types of nodes.

- Input Node : Each variable present in the set of assertions to be simulated is assigned an input node. Input nodes have zero in-degree.
- Internal Node : Each internal node is associated with a subexpression, which consists of only a single type of operator and one or more operands. The internal node holds the evaluated value of the subexpression. Internal nodes have a non-zero in-degree as well as a non-zero out-degree.
- Assertion Node : Each assertion node corresponds to a particular assertion and holds the result of that assertion across cycles. Assertion nodes have non-zero in-degree and zero out-degree.

The level of a node is defined as follows:

Level of input node = 0

Level of other nodes = max(level of the immediate predecessor of the node) + 1

Edges : There are two types of edges.

- Strong Edge : A directed edge between two nodes which causes the destination node to be assigned a fixed value that does not change in the future clock cycles is termed as a strong edge, represented by thick lines in our figures. Once a strong edge has been traversed, the destination node is never evaluated again in future. Figure1 (d) shows an example strong edge, where the source node contains ϕ and the destination node contains G. Intuitively, for a formula of the form $G\phi$, if ϕ evaluates to False (F) / undefined (U), the destination node (in this case, the G node) does not need to be evaluated again, since its status is completely determined. This is represented as an annotation on the edge. If ϕ evaluates to $True(T)$, this is not the case.
- Ordinary Edge : A directed edge that connects two nodes, where the value of the destination node may change across cycles. Figure1 (a) shows the example of an ordinary edge, shown as a dashed line.

Edges are annotated with a val_{in}/val_{out} notation where val_{in} and val_{out} take values from the set $\{True(T), False(F), Unknown(U)\}$. For example, in Figure1 (a), the edge is annotated by T/F and F/T. The markings signify that if the source node has value val_{in}, then val_{out} is propagated to the destination node. There can be one or more marking corresponding to an edge, depending on the status of the source node.

Having the nodes, the edges and their annotations as above, we now present some simple rules for creating an equivalent graph representation of the LTL expressions below. We define *path* as a sequence of truth values spread across consecutive cycles, corresponding to one evaluation result of the assertions.

Rule 1: ***Operator NOT*** ***(¬)***: For an expression $\neg\phi$, ϕ being a proposition, the corresponding graph is shown in Figure 1(a). If the source node is T (F), then the destination node will be set to F (T). Hence the edge markings are T/F and F/T.

Rule 2: ***Operator AND*** ***(∧)***: For the expression $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$, the corresponding graph is shown in Figure 1(b). If any one of the operands is $False$, then the result is $False$, irrespective of the value of the other operands. Hence the edges from source to destination are marked by F/F. If the source is $Unknown$ (which may happen for temporal expressions), the destination node can be $Unknown$ and hence the edges have the marking U/U as well. The default value of the

destination node in this case is set to $True$ because it implies that none of the operands were $False$ / $Unknown$, otherwise the node would have been set to $False$ / $Unknown$ by the corresponding source node. The destination node will be set by a value only when it holds $True$ / $Unknown$ and a different value is propagated through the edge. For example, if the destination node is currently holding a value $True$, it will be overwritten with the value $False$ / $Unknown$ depending on what value appears on the edges. It is to be noted that the value $True$ is never propagated in case of the AND operator, as apparent from Figure 1(b) edge markings. These rules are summarized in Table I a) where EV is the Evaluated Value sent via the edge and NV is the source node's existing value.

EV/NV	T	F	U
F	F	F	F
U	U	F	U

a) AND

EV/NV	T	F	U
T	T	T	U
U	U	U	U

b) OR

TABLE I: Rules for setting operator node values

Rule 3: ***Operator OR*** ***($\vee$)***: For the expression, $\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n$, the corresponding graph is shown in Figure 1(c). If any of the operands is $True$, then the result is $True$, as per the semantics of the OR operator. Hence the edges from operator node to operand node are marked by T/T. If the operand is $Unknown$, then the operator node will be $Unknown$ and hence the edges have the marking U/U as well. The default value of the operator node is $False$ because it implies that none of the operands were $True$ / $Unknown$. The rules for evaluation of the OR operator node is presented in Table I (b).
Rule 4: ***Operator GLOBAL*** ***(G)***: For the expression $G\phi$, the corresponding graph structure is shown in Figure 1(d). The GLOBAL operator signifies that ϕ has to hold on the entire path from the current cycle onwards. Hence if ϕ is $False$ or $Unknown$ in a given clock cycle, the expression will be $False$ all through from that clock cycle. The source to destination is thereby connected by a strong edge marked F/F and U/F. The default value for the G operator node is $Unknown$.
Rule 5: ***Operator EVENTUALLY*** ***($\mathcal{F}$)***: For the expression $\mathcal{F}\phi$, the corresponding graph structure is shown in Figure 1(e). The EVENTUALLY operator signifies that ϕ eventually has to hold. Once ϕ is $True$, the expression will be $True$ all through from that clock cycle, and hence the operator to operand is connected by a strong edge marked T/T. The default value for the $\mathcal{F}$ operator node is $Unknown$.
Rule 6: ***Operator UNTIL*** ***($\mathcal{U}$)***: The graph structure corresponding to $\phi_1\ \mathcal{U}\ \phi_2$ is shown in Figure 2(a). The UNTIL operator signifies ϕ_1 has to hold at least until ϕ_2, which holds at the current or a future position. If the internal $\vee$ node is $False$, from the definition of UNTIL, the $\mathcal{U}$ operator node is set to $False$ from the current cycle. On the other hand, if ϕ_2 is $True$, the UNTIL operator node will be $True$ from the current cycle onwards. The default value here is $Unknown$.
Rule 7: ***Operator NEXT*** ***(X)***: We do not explicitly create a node for the NEXT operator, rather at each node, we store a delay t_d of that node as an attribute of that node. We use

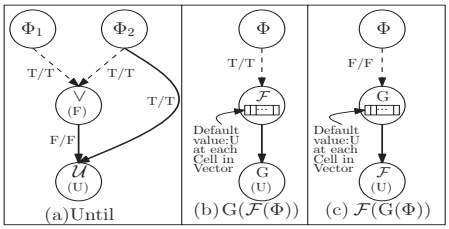

Fig. 2: *Graph Structure of Until and nested temporal operators*

the distributive property of the next operator, i.e., $X(\Phi_1 \vee \Phi_2) = X(\Phi_1) \vee X(\Phi_2)$; $X(\Phi_1 \wedge \Phi_2) = X(\Phi_1) \wedge X(\Phi_2)$; $X(\neg(\Phi_1) = \neg(X(\Phi_1))$ to associate the next operator with the variable itself, instead of associating it with an expression. Each assertion is transformed into our desired form. We now introduce the notion of the 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.

Nested temporal operators require special handling during simulation. We explain below two specific cases.

- $G(\mathcal{F}(\phi))$: The corresponding graph structure is shown in Figure 2(b). Instead of a single value, the $\mathcal{F}$ operator node stores a vector of truth values. Each value in the vector corresponds to the value of an instance of the $\mathcal{F}$ operator starting at each new cycle. Whenever the propositional expression ϕ evaluates to $True(T)$, the corresponding value in the vector is set to T. At the end of simulation, the value of the G operator node is set to T if all the values in the F operator node value vector are T, otherwise to $False(F)$ since it implies that at least one instance of F operator evaluated to either $Unknown(U)$ or $False(F)$.
- $\mathcal{F}(G(\phi))$: The treatment of $\mathcal{F}(G(\phi))$ is similar. The corresponding graph structure is shown in Figure2(c). At the end of simulation, the value of the G operator node is set to T if at least one value in the value vector of the G operator is not $False(F)$.

A. Pre-processing assertions to generate Look Up Table

The preprocessing stage of the methodology is executed only once for a set of assertions and generates a look up table corresponding to the assertions. We explain below what a look up table is and how we obtain it from the graph structures presented earlier. For each source node, we store a list of destination nodes that need to be considered for this source node with value annotations as appropriate. For an edge n_i to n_j with edge marking val_{in}/val_{out}, we store node n_j with val_{out} in the associated list (also called the val_{in}) list of node n_i. In addition, the type of the edge is also saved in the entry corresponding to n_j in the same list of n_i. Thus, during simulation, if node n_i has value val_{in}, we can set the value of the successor nodes of n_i by looking up the val_{in} list of node n_i that we created in the preprocessing step. We

define this set list structures as the Look Up Table (LUT). It is to be noted that such a look up table captures all the elements of the corresponding graph data structures.

Initially the assertions are processed and the look up table corresponding to the set of assertions is generated. We explain the preprocessing stage using a detailed example on the set of assertions presented in Section II.

Example 1. *Consider the set of assertions below.*
[P_1] : $G(a \vee c)$; [P_2] : $G(a \wedge Xb)$; [P_3] : $\mathcal{F}(c \vee d \vee e)$
The set of assertions converted to postfix is :-
[P_1] : $ac \vee G$; [P_2] : $abX \wedge G$; [P_3] : $cd \vee e \vee \mathcal{F}$

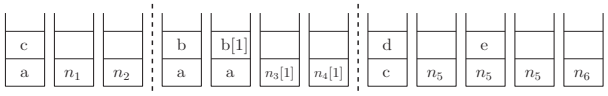

Fig. 3: *Preprocessing the assertion set*

The input nodes a, b, c, d and e are at level 0 from the definition of level. For assertion P_1, a and c are pushed to the stack. Thereafter, the $\vee$ operator is encountered and hence a and c are popped from stack. We create a new operator node n_1 corresponding to this operator with default value $False$ as per Rule 3 and add it to the T and U lists of a and c respectively. Since a and c are at level 0, level of n_1 is set to 1 as per the definition of level. Then, n_1 is pushed into the stack. We encounter the G operator and pop n_1 from the stack. A G operator node n_2 is generated with default value U and n_2 is added to the $False$ list of n_1 as per Rule 4 with a strong edge connection. This concludes preprocessing of assertion P_1. Assertion 2 presents an interesting case due to the presence of the X operator before operand b. Operands a and b are pushed to the stack. On encountering the X operator, b is marked with delay equal to 1 as per Rule 7 without creating any additional node. For the $\wedge$ operator, a new $\wedge$ operator node n_3 with default value T is created and it is added to the $False$ and $Unknown$ lists of a as well as that of c according to Rule 2. The delay of n_3 is set to 1 since the maximum delay of its immediate predecessors is 1. The G operator node n_4 is generated with default value U and n_4 is added to the $False$ list of n_3 as per Rule 4 with a strong edge marking. n_4 is also assigned a delay 1, as per our level assignment strategy. For assertion 3, we proceed similarly. For the second $\vee$ operator in P_3 that we encounter, we do not create an additional node, rather we place it in the $True$ and $Unknown$ lists of input node e, the previously created $\vee$ operator node n_5. Table II shows the look up table generated after completion of the preprocessing stage. In the table, $n[s]$ (e.g., $n_2[s]$) represents that the value (T / F) is propagated to the specific node n through a strong edge, and thereby the value is never changed. For ease of visualization, we show the shared graph structure in Figure4.

The generated look up table is used as input to the simulation stage described in Table II.

Level	Type	Node	List type	List
0	Input	a	True False Unknown	n_1 n_3 n_1,n_3
0	Input	b[1]	False	n_3
0	Input	c	True Unknown	n_1,n_5 n_1,n_5
0	Input	d	True Unknown	n_5 n_5
0	Input	e	True Unknown	n_5 n_5
1	$\vee$	n_1	False	n_2[s]
1	$\wedge$	n_3[1]	False	n_4[s]
1	$\vee$	n_5	True	n_6[s]
2	G	n_2	-	-
2	G	n_4[1]	-	-
2	$\mathcal{F}$	n_6	-	-

TABLE II: Look Up Table

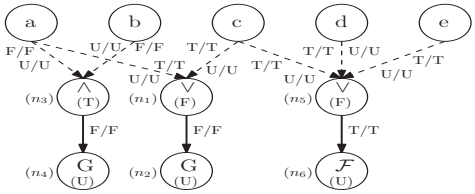

Fig. 4: *Shared Graph Data Structure*

B. Assertion simulation using Look Up Tables

In this subsection, we discuss our thread based assertion simulation strategy. Nodes that are in assertions without any temporal operators, are evaluated using value substitution at the start of simulation and results are reported. Otherwise, in each cycle, a new thread is created for the evaluation. If the evaluation of an assertion starts at clock cycle t_0, each node n_i at delay t_d is evaluated at the $(t_0 + t_d)^{th}$ cycle. The maximum number of threads that can be active in memory simultaneously is equal to the maximum delay of any node in the shared data structure. We now present a level-based assertion simulation strategy. This method arranges the nodes in order of the level to which it belongs. The algorithm proceeds by initializing the nodes to their default values. Thereafter, it starts a new thread for simulation, say at clock cycle t_0. At clock cycle $t_i (>= t_0)$, the thread reads the simulation inputs for input nodes with temporal delay $(t_i - t_0)$ and then processes nodes in increasing order of level. Processing involves checking the current value val_{curr} (T / F / U) of the node n_i and setting the values of the nodes in the corresponding list of the node n_i as saved in the lookup table. Once all the nodes in the current level have been processed, the nodes in the next level are taken up. If an internal node has not been set to a value, it stays at its default value. To take into account the delay of the nodes, the simulation thread only starts by processing the input nodes at the current delay and waits across cycles to process the other input nodes at greater delays. The assertion nodes are set to their default value at start of simulation and shared amongst

all the threads so that there is a consistent visible result of assertion evaluation. To do so, assignment of truth value to assertion nodes by threads is done in a thread safe manner, keeping in mind the execution semantics of LTL operators. Algorithm 1 presents the detailed methodology of the work done by a single thread. Each thread reads the simulation values and handles the evaluation mechanism based on the look up table up to a maximum delay of any assertion.

Algorithm 1 Algorithm for level based assertion simulation

Input: Look-up table, max_level, max_temporal_depth in
Output: out

1: *Initialization* : Set each $node$ to default value according to the $type$ of node.
2: *Start a new thread for evaluation*
3: **for** $time_{curr} = 0$ to max_delay **do**
4: Read simulation inputs to input nodes with temporal delay $time_{curr}$
5: **for** $level = 0$ to max_level **do**
6: **for** $node_i \in level$ and $(node_i[time] == time_{curr}$ or $isEvaluated(node_i) == True)$ **do**
7: **if** $(node_i == True)$ **then**
8: Set values to nodes in $True$ list of $node_i$
9: **else if** $(node_i == False)$ **then**
10: Set values to nodes in $False$ list of $node_i$
11: **else**
12: Set values to nodes in $Unknown$ list of $node_i$
13: **end if**
14: Set evaluated flag for the new value assigned nodes to $True$
15: **end for**
16: **end for**
17: *Wait till next clock cycle*
18: **end for**

We explain the working of our algorithm on Example 1. Table III (a) shows the order of evaluation. We explain the demonstration of simulation shown in Table IV, using simulation inputs stated in Table III (b).

Level	Nodes[0]	Nodes[1]
0	a c d e	b
1	n1	n5 n3
2	n2	n6 n4

a) Nodes visualized level-wise

Time	a	b	c	d	e
cc_0	T	F	T	F	T
cc_1	T	T	T	T	T
cc_2	T	F	T	F	T
...	...				

b) Simulation inputs

TABLE III: Node Evaluation Order

cc_0	cc_1	cc_2
T_0		
n_1 = True n_5 = True	n_3 = True	
n_2 = U n_6 = True	n_4 = U	
	T_1	
	n_1 = True n_5 = True	n_3 = False
	n_2 = U	n_4 = False

TABLE IV: Simulation of assertions

- In clock cycle 0 (cc_0), Thread 0 (T_0) starts execution. We note that the maximum delay is 1 (due to P_2), and hence each thread will be alive for two clock cycles. T_0 reads the inputs a, c, d, e needed for this clock. As per the semantics of our level order simulation strategy, b is not present at level 0 and hence not read. As a is $True$, n_1 is assigned $True$. Similarly, since c is $True$, n_5 is assigned $True$. Since a is $True$ but b is not known, no value is assigned to node n_3 which is at delay 1. Thereafter, as n_5 is True, n_6 is assigned $True$ via a strong edge and hence n_6 is eliminated and reported as $True$. n_2 stays at assigned default value U. Thus we can observe that the algorithm proceeds level wise to assign the values to nodes in the next level depending on their present value.
- In clock cycle 1 (cc_1), T_0 reads the value of b which is True and it does not assign a value to n_3. n_3 stays at the default value $True$. Similarly, n_4 stays at its default value U. T_0 terminates at the end of this cycle. In this clock cycle, a new thread T_1 starts execution. Similar to T_0 at cc_0, T_1 assigns both n_1 and n_5 to $True$ at this cycle. However, now n_5 does not assign any value to n_6 since the strong edge was already traversed by T_0 in cc_0. n_2 stays at its default value U.
- In clock cycle 2 (cc_2), T_1 reads b which is $False$ and thereby sets n_3 as $False$, which in turn sets n_4 to $False$. n_4 is eliminated and $False$ is reported as the final value of n_4. T_1 terminates. Another new thread will begin and proceed as mentioned above.

The values of the assertions after 3 cycles of simulation (cc_0, cc_1, cc_2) are available as the assigned values of the nodes n_2, n_4 and n_6 as done above.

C. Optimizations

We propose two optimizations to speed up our methodology.

- Once an assertion with a G ($\mathcal{F}$) operator has been evaluated to F (T), it will not be affected by the simulation inputs in the future cycles. Thus the inputs that relate to only these assertions need not be considered as well in the future cycles, provided these inputs are not relevant for other assertions. This can be achieved by using an *outdegree* field in the input nodes. In the assertion nodes that do not change values once evaluated, the list of inputs (*inputlist*) driving the assertion is stored. Once the assertion has been evaluated, we decrement the *outdegree* of the nodes in the *inputlist* by one. If any of the input node has *outdegree* zero, it is not considered in future.
- For common subexpressions across assertions, we have duplicate nodes in the same level. During preprocessing, we can merge nodes with identical immediate predecessors and edge marking in the same level, reducing the number of lookups using these common subexpressions.

IV. Implementation and Results

The proposed LTL simulation architecture is shown in Figure 5. The input LTL assertions are fed to the preprocessing stage (implemented in Python). The assertions are passed on to the LUT Generator which generates the LUT representation of the assertions. For simulating the same assertion set, the prepro-

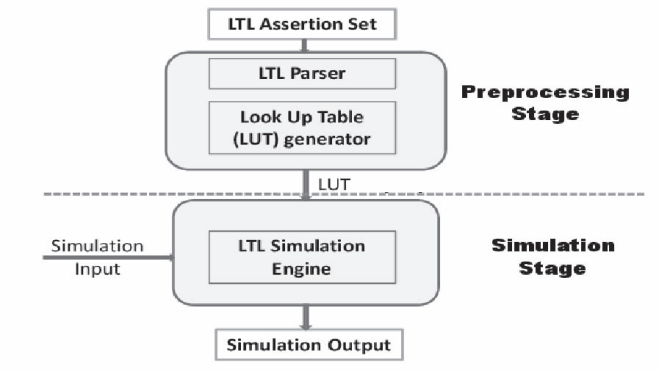

Fig. 5: *Proposed LTL Simulation Architecture*

cessing stage needs to be executed only once for generating the LUT. The LTL assertion evaluation engine, implemented in Java, takes the LUT generated from the preprocessing stage as input along with the simulation inputs. This engine is to be interfaced with actual simulators. For the purpose of this experiment, we developed a simple cycle based simulator which takes in the simulation inputs for each clock cycle and executes our level based assertion evaluation algorithm.

We put to test our proposed Level Based Simulator (LBS) without the optimizations against an in-house LTL simulator (LSim) that implements assertions as individual monitors and evaluates them with rudimentary simulation. Randomly generated simulation inputs, assertions and number of simulation cycles were used to compare the relative performances. The statistics, shown in Table V demonstrate the considerable gains of LBS in runtime (both recorded as seconds in Columns 5 and 6) over a classical evaluation mechanism. The memory requirements were in the order of Kilo-bytes for both. All experiments were performed on an Intel Core 2 Duo P8700 (2.53 GHz) with 3 GB RAM. As evident from the results, the LUT-based evaluation mechanism made significant performance difference. LBS was also evaluated on a subset of

Exp	#Assertions	#Vars	#Sim. cycles	LBS	LSim
1	31	4	753	0.083	38.847
2	26	6	899	0.125	114.687
3	33	6	1159	0.154	148.99
4	15	18	1690	0.181	177.29

TABLE V: Performance comparison of LBS vs LSim

the assertions from a commercial SVA-based assertion IP for the Open Cores Protocol (OCP) [1] with a rudimentary OCP design model, and assertions written in LTL. Table VI shows the results on different number of simulation iterations on random test-benches. The average runtime and peak memory requirements of the same OCP design on the same random test-bench for commercial ABV simulators like VCS [5] and Questa [3] were in the order of 10s, however the peak memory requirement was an order of magnitude greater than ours. In fact, for the final run, both VCS and Questa ran out of memory and got killed. We do not furnish comparative results here since the high memory requirement may as well be attributed to the simulation data structure overhead and we could not find a direct way to get the peak memory required for assertion handling only. We are currently working on integrating our assertion evaluation strategy inside these ABV engines and comparing the run-time and memory.

	No of signals	46
	No of assertions	45
Look up Table Generation Time		0.455s
No of cycles simulated	**Simulation time (in secs)**	**Peak memory (in mb)**
100	0.763	24.28
1000	0.2888	12.4906
10000	1.5862	87.4856
100000	11.7932	154.49
1000000	117.2896	169.992

TABLE VI: LBS performance for OCP assertions

V. Conclusion and Future Work

In this paper, we presented a new methodology for simulation of LTL assertions. The foundation of the work is based on inferring the evaluation results of the assertions without actual evaluation by using a shared data structure implemented as a look up table. We are currently working on integrating our evaluation framework with commercial ABV simulators.

Acknowledgment

This work is supported, in part, by a special PPEC-grant (2012-2017) to Nanotechnology Research Triangle provided by Indian Statistical Institute, Kolkata. The authors would also like to acknowledge Interra Systems India Pvt. Ltd. for donating their OCP VIP for this experimentation.

References

[1] Open Core Protocol. accellera.org/downloads/standards/ocp/.
[2] PSL. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4408637.
[3] Questa Advanced Simulator. www.mentor.com/products/fv/questa/.
[4] SVA. http://standards.ieee.org/getieee/1800/download/1800-2012.pdf.
[5] Synopsys VCS Simulator. http://www.synopsys.com/tools/verification/functionalverification/\pages/vcs.aspx.
[6] Koji Ara and Kei Suzuki. A proposal for transaction-level verification with component wrapper language. In *DATE: Designers' Forum-Volume 2*, page 20082. IEEE Computer Society, 2003.
[7] Armoni et al. The forspec temporal logic: A new temporal property-specification language. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 296–311. Springer, 2002.
[8] Armoni et al. Deterministic dynamic monitors for linear-time assertions. In *Formal Approaches to Software Testing and Runtime Verification*, pages 163–177. 2006.
[9] Chang et al. A simulation-based temporal assertion checker for psl. In *Proceedings of IEEE International Symposium on Micro-NanoMechatronics and Human Science*, pages 1528–1531, 2003.
[10] Ajay J Daga et al. A symbolic-simulation approach to the timing verification of interacting fsms. In *ICCD*, pages 584–584, 1995.
[11] Maksim Jenihhin et al. Psl assertion checking with temporally extended high-level decision diagrams. In *Proc. IEEE Latin-American Test Workshop*, pages 49–54, 2008.
[12] G. Pinter et al. Automatic generation of executable assertions for runtime checking temporal requirements. In *HASE*, pages 111–120, 2005.
[13] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.
[14] Tabakov et al. Optimized temporal monitors for systemc. *Formal Methods in System Design*, pages 236–268, 2012.