TABLE I
RESULTS FOR KNOWN PATTERNS OF FINITE SIZE

Pattern Name	Area Size (Pixel)	# of Tiles in Trivial Set	# of Tiles in Synthesized Set	Reduction Percentage
Sierpinski	6×6	36	21	42%
	8×8	64	37	43%
	12×12	144	80	45%
	17×17	289	172	41%
	32×32	1024	595	42%
Line1	6×6	36	2	95%
	15×15	225	2	99.2%
Line2	6×6	36	19	47%
	15×15	225	133	41%
Binary Counter	6×6	36	19	47%
	15×15	225	121	46%
Chess-Board	6×6	36	19	47%
	15×15	225	119	47%

self-assembly for nanomanufacturing. In the second set of experiments, the Sierpinski triangle, bar code (including line1 and line2), chessboard, and binary counter have been selected as patterns. Theoretically, these types of pattern can grow infinitely based on self-assembly rules. Synthesis, however, can only be applicable to a finite pattern. Therefore, in this set of experiments, a limited pixel area of the pattern is selected and provided as input to the program.

Table I shows the results, inclusive of the selected pixel area size, the number of tiles in the trivial tile set, the number of tiles in the valid tile set (as obtained by synthesis), and the percentage reduction in tile types. For a pattern (i.e., line1), the program produces the optimal tile set. However, for most of the patterns, the reduction in tile types is similar in performance to random tiles.

VI. CONCLUSION

In this paper, the synthesis problem for generating a tile set for a finite pattern in DNA self-assembly has been presented as a combinatorial optimization problem referred to as the PATS problem. A graph model has been proposed for analyzing the tile sets assembling a specified patterns. The PATS problem has been analyzed by utilizing the proposed graph model. Two greedy algorithms (referred to as PATS_Tile and PATS_Bond) have been proposed for the PATS problem. Both algorithms are greedy and have the same execution complexity $O(l^4)$ for a square pattern of size $l \times l$. Self-assembly from the tile sets synthesized by the proposed algorithms was verified by the Xgrow simulation. The PATS problem and the proposed algorithms only utilize the rule tiles in the tile set. The design of seed and boundary tiles also needs to be addressed in future research to fully characterize the self-assembly process. Moreover, the tile synthesis problem analyzed in this paper relies on aTAM as model. aTAM has been extensively analyzed in the technical literature; it is however important to assess whether the assumptions of aTAM (for example, the nonrotational property of the tiles) limit its widespread application and, if required, its modification to account for other self-assembly scenarios as encountered in different chemical and biological processes (for example, the case in which horizontal and vertical bonds are correlated).

REFERENCES

[1] R. Compano, L. Molenkamp, and D. Paul, "Technology roadmap for nanoelectronics," *Eur. Commission IST programme, Future and Emerging Technologies*, 2000. [Online]. Available: cordis.europa.eu/ist/fet/nidqf.htm

[2] C. Lin, Y. Liu, S. Rinker, and H. Yan, "DNA tile based self-assembly: Building complex nanoarchitectures," *ChemPhysChem*, vol. 7, no. 8, pp. 1641–1647, 2006.

[3] S.-H. Park, R. Barish, H. Li, J. Reif, G. Finkelstein, H. Yan, and T. LaBean, "Three-helix bundle DNA tiles self-assemble into 2D lattice or 1D templates for silver nanowires," *Nano Lett.*, vol. 5, no. 4, pp. 693–696, 2005.

[4] P. W. K. Rothemund, "Folding DNA to create nanoscale shapes and patterns," *Nature*, vol. 440, no. 7082, pp. 297–302, Mar. 2006.

[5] S. H. Park, C. Pistol, S. J. Ahn, J. H. Reif, A. R. Lebeck, C. Dwyer, and T. H. LaBean, "Finite-size, fully-addressable DNA tile lattices formed by hierarchical assembly procedures," *Angew. Chem., Int. ed. Engl.*, vol. 45, no. 5, pp. 735–739, Jan. 2006.

[6] W. Hu, K. Sarveswaran, M. Lieberman, and G. Bernstein, "High-resolution electron beam lithography and DNA nano-patterning for molecular QCA," *IEEE Trans. Nanotechnol.*, vol. 5, no. 3, pp. 312–316, May 2005.

[7] C. Mao, W. Sun, and N. C. Seeman, "Designed two-dimensional DNA Holliday junction arrays visualized by atomic force microscopy," *J. Amer. Chem. Soc.*, vol. 121, no. 23, pp. 5437–5443, 1999.

[8] E. Winfree, F. Liu, L. A. Wenzler, and N. C. Seeman, "Design and self-assembly of two-dimensional DNA crystals," *Nature*, vol. 394, no. 6693, pp. 539–544, Aug. 1998.

[9] C. Mao, T. H. LaBean, J. H. Reif, and N. C. Seeman, "Logical computation using algorithmic self-assembly of DNA triple-crossover molecules," *Nature*, vol. 407, no. 6803, pp. 493–496, Sep. 2000.

[10] S.-H. Park, H. Yan, J. Reif, T. LaBean, and G. Finkelstein, "Electronic nanostructures templated on self-assembled DNA scaffolds," *Nanotechnology*, vol. 15, no. 10, pp. S525–S527, Oct. 2004.

[11] H. Yan, S. H. Park, G. Finkelstein, J. H. Reif, and T. H. LaBean, "DNA-templated self-assembly of protein arrays and highly conductive nanowires," *Science*, vol. 301, no. 5641, pp. 1882–1884, Sep. 2003.

[12] J. Lund, J. C. Dong, Z. X. Deng, C. D. Mao, and B. A. Parviz, "Electrical conduction in 7 nm wires constructed on λ-DNA," *Nanotechnology*, vol. 17, no. 11, pp. 2752–2757, Jun. 2006.

[13] E. Winfree, *Xgrow homepage*. [Online]. Available: www.dna.caltech.edu/Xgrow/

[14] P. W. K. Rothemund, N. Papadakis, and E. Winfree, "Algorithmic self-assembly of DNA Sierpinski triangles," *PLoS Biology*, vol. 2, no. 12, p. e424, Dec. 2004. DOI: 10.1371/journal.pbio.0020424.

[15] E. Winfree and R. Bekbolatov, "Proofreading tile sets: Error correction for algorithmic self-assembly," in *Proc. 9th Int. Workshop DNA Comput.*, 2003, pp. 108–126.

[16] X. Ma, *On the complexity of DNA self-assembly processes*, 2007. Internal Report, NEU, available upon request.

Accelerating Assertion Coverage With Adaptive Testbenches

Bhaskar Pal, Ansuman Banerjee, Arnab Sinha, and Pallab Dasgupta

***Abstract*—We present a new approach to bias random test generation for accelerating assertion coverage. The novelty of the proposed approach is that it treats the design under test as a black box and attempts to steer the simulation toward coverage points that are relevant for targeted assertions purely through external control. We present this approach over three different models with varying degrees of observability and control. The results demonstrate a significant speedup in assertion coverage as compared to randomized simulation.**

***Index Terms*—Design verification, functional coverage, test generation.**

Manuscript received June 16, 2007; revised October 31, 2007. The work of P. Dasgupta was supported in part by the Department of Science and Technology, Government of India. This paper was recommended by Associate Editor R. F. Damiano.

The authors are with the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur 721302, India (e-mail: pallab@cse.iitkgp.ernet.in).

Digital Object Identifier 10.1109/TCAD.2008.917975

0278-0070/$25.00 © 2008 IEEE

Authorized licensed use limited to: Princeton University. Downloaded on August 19, 2009 at 15:33 from IEEE Xplore. Restrictions apply.

I. Introduction

Capacity limitations continue to impede widespread adoption of formal property verification technology, but assertions are widely used in dynamic property verification (DPV), where assertions are monitored over simulation traces on traditional simulation-based verification environments.

DPV is essentially bug hunting, i.e., the goal is not to prove a property formally but to keep looking for bugs over simulation traces. Therefore, an assertion is not covered unless the testbench drives the simulation to those scenarios (typically corner-case scenarios) for which the property was developed. Writing directed tests for covering specific assertions is time-consuming and goes against the industry trend of developing constrained random testbench architectures, primarily to avoid the task of writing thousands of directed tests. In this paper, we present a technique for biasing a randomized testbench with respect to a given property (one at a time) so that it reaches the scenarios that are relevant to that property in less time.

In our model, we choose to treat the design under test (DUT) as a black box and formulate the problem as a game between the DUT and the testbench, where the testbench wins when the simulation reaches a scenario relevant to the given property. Our decision to treat the DUT as a black box is motivated by two factors.

1) Our method is not affected by the implementation of the DUT. For example, the DUT may be in a nonsynthesizable language.
2) The size of the DUT is not an issue for us as long as it can be handled by the simulation tool.

Our approach is a novel alternative to existing test-generation methods that parse the DUT to trace a path to a given scenario. Methods such as property-specific test generation [1], counter-example-guided test generation [2], [3], sequential automatic test pattern generation [4], [5], and model-based test generation [6] use DUT-specific information to reach corner-case scenarios quickly. On the other hand, the task of analyzing the DUT leads to capacity issues and restrictions on the language used for the DUT. Our goal is not to compete with these methods in the space where they work but to present an alternative technology that is easily scalable, can be integrated easily with other automated approaches (which utilize design data) and existing simulation platforms [7], and can handle a DUT in any language that is supported by the simulation environment.

Since we choose to treat the DUT as a black box, our algorithms for test generation vary with two factors:

1) observability: the DUT signals that are visible to the test generator during simulation;
2) controllability: the DUT signals that can be controlled directly/indirectly by the test generator.

We present algorithms for three different models in the next three sections, which differ in terms of observability and controllability. We demonstrate the result for all the three models in Section V.

II. Model 1: Direct Vacuity

In the direct vacuity model, we assume that the assertions are defined over the signals appearing at the interface between the testbench and the DUT. In other words, each signal of the targeted assertion is observable, and the inputs to the DUT are controllable. The goal is to decide the inputs at each time step so that the trace satisfies the assertion nonvacuously.

The definition of vacuity is not uniform, and verification engineers use various metrics for assertion coverage. In many tools, an assertion in implication form is said to be covered whenever the antecedent part of the implication matches. For example, consider the following property (in SVA [12]), for an arbiter with input r and output g, which says that if the request r is asserted, then the grant g must be asserted in the next two cycles, unless r is lowered in between

```
property P;
@(posedge clk) r |-> ##1 (g or (!g && !r) or
                          ((!g && r) ##1 g));
endproperty
```

Obviously, the above assertion is vacuously satisfied in all traces where r is never asserted. Some tools will report the assertion to be covered each time r is asserted, because r satisfies the antecedent of the implication. This is not a formal definition of vacuity since we could write the same property without using the implication—in which case, the context of the property would become implicit.

Implication vacuity also overlooks a very important fact which is demonstrated by the scenario when the testbench drives r at cycle t but does not get the grant g in the next cycle, $t+1$. In this case, the testbench must drive r again to test whether the arbiter asserts g at $t+2$. On the other hand, if the testbench lowers r at $t+1$, then the property will be satisfied vacuously. In other words, the coverage of a temporal property (like P) may continue over many cycles, and we have a coverage hit only when the response of the DUT eventually decides the truth of the property.

Driving a nonvacuous scenario may therefore be seen as a game between the DUT and its testbench. In each cycle (round of the game), the testbench must choose the values of the inputs to the DUT in such a way that the truth of the assertion is not solely determined by the values of the input signals but depends on the response of the DUT in the next cycle. The property is satisfied nonvacuously if the response of the DUT eventually decides the truth of the assertion. In this paper, we shall refer to this game as the vacuity game. We introduced the first formal methods on direct vacuity games (DVGs) in [8].

In all forms of vacuity games, we must realize that the testbench has to drive many other signals (in a protocol-compliant manner) that do not appear in a given assertion—a task which is hard to automate. We therefore propose to embed our methods within an existing testbench architecture so that the testbench queries our methods in each cycle for only the input values that are relevant to the target assertion. Our methods randomly choose (to achieve proper distribution of stimulus) among the input valuations that are relevant in that cycle. This way, we have integrated our methods with existing layered testbench architectures and have achieved considerable success.

A. Formal Model

Formally, we define a module J as a design block having a set of inputs I, a set of outputs O, an initial block INIT, and a description $\mathcal{B}$. The execution of the initial block produces the values of the output variables at the beginning of the simulation. The formal properties are specified in linear temporal logic (LTL). A detailed description of this formal model can be found in [8].

Definition 1: X-guarded LTL formula. An LTL property is X-guarded if all its terms start with the X operator. □

For example, the property $\mathcal{P} = ((X\ p)U(X\ X\ q)) \wedge (X\ F\ r)$ is an X-guarded formula since the X-pushed form of $\mathcal{P}$, $\mathcal{P}_X = X((p\ U(X\ q)) \wedge (F\ r))$, starts with an X operator whose scope covers the whole formula.

The task of monitoring the truth of a given LTL property along a simulation run works as follows. If we are required to check an LTL property φ from a given time step t, we rewrite the LTL property into a set of propositions over the signal values at time t and a set of X-guarded LTL properties over the run starting from time $t+1$.

The property checker reads the signal values from the simulation at time t and substitutes these on the rewritten properties and derives a

Authorized licensed use limited to: Princeton University. Downloaded on August 19, 2009 at 15:33 from IEEE Xplore. Restrictions apply.

new property that must hold on the run starting from $t+1$ by dropping the leftmost X operator from each X-guarded term.

To check the property $p\,U(q\,U\,r)$ at time t, we rewrite it as

$$(r\ \vee\ (q\ \wedge\ X(q\,U\,r)))\ \vee\ (p\ \wedge\ X\,(p\,U\,(q\,U\,r)))\,.$$

If the simulation at time t gives $p=0$, $q=1$, and $r=0$, then by substituting these values, we obtain the property $X(q\,U\,r)$. Therefore, at time $t+1$, we need to check the property $q\,U\,r$. We repeat the same methodology on $q\,U\,r$ at time $t+1$.

For automatic test generation, we may choose the values of the input signals at each time step t while monitoring the property. The following definition is useful to characterize our goal.

Definition 2: Vacuous input vector. An input vector $\widehat{I}$ is vacuous at a given state with respect to a property φ iff φ becomes true at that state on input $\widehat{I}$ regardless of the values of the remaining variables. □

For each property, our target is to drive a sequence of inputs such that a nonvacuous success/failure of the property is reported.

B. Test-Generation Algorithm

To develop the formal algorithms, let us study the interaction between the DUT J and its testbench with respect to a given property L. The execution of the INIT block is the first move of the DUT. If the initial state is sufficient to satisfy or refute L, then we have a nonvacuous success or failure. Otherwise, there must exist some nonvacuous input vectors with respect to L. The test generator must choose one such assignment and simulate J with that input and study the response of J (that is, the values of the outputs) in the next cycle. If L gets satisfied or refuted, then we have a nonvacuous success or failure; else, we repeat the same in the next cycle.

The procedure SimulateMain outlines our algorithm for intelligent test generation for a module J and a target-uncovered property L. It calls procedure GenStimulus to produce a nonvacuous input vector with respect to L. For the input variables occurring in L, it chooses a nonvacuous assignment. For the rest of the input variables, a random assignment is made since their valuations do not affect the vacuity of the property.

Algo. 2.1: **Procedure SimulateMain**
InputVector $\widehat{I}$ SimulateMain(Output: $\widehat{O}$, property: L)
Step 1: Rewrite L in terms of present state Boolean propositions and X-guarded temporal properties
Step 2: Substitute the values of the output variables from $\widehat{O}$ in the non X-guarded terms of L to obtain $\widehat{L}$
Step 3: If $\widehat{L}=$ TRUE, return with nonvacuous success
Step 4: If $\widehat{L}=$ FALSE, return with nonvacuous failure
Step 5: $\widehat{I}=$ GenStimulus($\widehat{L}$)
Step 6: Obtain L' from $\widehat{L}$ by substituting $\widehat{I}$ in the non X-guarded terms of $\widehat{L}$ and dropping the leftmost X from each X-guarded temporal property
Step 7: Set $L=L'$
Step 8: Return $\widehat{I}$

Algo. 2.2: **Procedure GenStimulus**
Input_Vector GenStimulus(property: L)
//L is a property over I & X-guarded terms over $I\bigcup O$
Step 1: Rewrite L as a conjunction of clauses, where each clause is a disjunction of Boolean formulas and X-guarded terms
Step 2: Set $P=$ the Boolean formula obtained from L after dropping the X-guarded terms
Step 3: If P is satisfiable, $\widehat{I}=$ random input vector refuting P else $\widehat{I}=$ any random input vector
Step 4: Return $\widehat{I}$

Example 1: Consider the LTL property below for an arbiter

$$P:\quad G\,(r\Rightarrow X\,(g\vee(r\Rightarrow X(g))))$$

with input r and output g. Initially, g is low. With this value, P is neither true nor false. GenStimulus is called with P, which is rewritten as

$$((\neg r\vee X\,(g\vee(r\Rightarrow Xg)))\wedge X(P))$$

as the argument, which returns a nonvacuous input vector $r=1$. The arbiter is simulated with this input vector. In response, suppose that the arbiter does not assert g in the next cycle. In this situation, GenStimulus is again called with the following argument:

$$((\neg r\vee X(g))\wedge(\neg r\vee X\,(g\vee(r\Rightarrow Xg)))\wedge X(P))\,.$$

GenStimulus returns a nonvacuous input vector $r=1$, and the arbiter is simulated. Now, the arbiter must assert g in the next cycle. □

III. Model 2: Indirect Vacuity

In the direct vacuity model, the targeted assertions are defined over signals that appear in the interface between the DUT and the testbench. However, in practice, verification engineers also write assertions to verify internal properties of the DUT over signals that are neither driven nor read by the testbench. For example, the top-level testbench for a processor typically drives a characteristic stream of instructions, but the verification task includes checking properties over internal events in the pipeline. Each instruction causes a known set of internal events in the pipeline, and only a specific sequence of instructions may lead the simulation toward specific scenarios (such as hazards) that are relevant to a given property of the pipeline. Our goal is to find such sequences of instructions on-the-fly during simulation.

Formally, let $H=\{h_1,h_2,\ldots,h_N\}$ be a set of instructions. An instruction h_i, which is driven at cycle t, causes a known set of events between cycle t and cycle $t+\delta_i$.

The events caused by an instruction are defined over a set of internal signals I. In our model, we assume that the exact sequence of events caused by an instruction is known *a priori*. However, the assertions to be checked are defined over $I\cup O$, where O is the set of other signals over which the testbench has no control. In other words, the valuation of O is decided by the DUT. Therefore, this is again a game that is similar to the one studied in the previous section, except that the values of the signals in I cannot be directly assigned by the testbench but may be indirectly controlled by driving appropriate instructions into the DUT. Since the values of O are decided by the DUT (which we choose to treat as a black box), the desired sequence of instructions has to be decided on-the-fly during simulation.

The events caused by h_i can be specified in terms of a formula

$$\bigwedge_{k=0}^{\delta_i} T_i^k$$

where T_i^k is a valuation of some signals in I at time $t+k$ when h_i is driven at time t. Since this specification is entered by the verification engineer, we need to check whether the specification is consistent before we use it in our test-generation algorithm.

Formally, for any two instructions h_i and h_j, if any of the following family of formulas are unsatisfiable, we reject the specification as inconsistent:

$$\exists a\exists b(0\le a\le\delta_i)\wedge(0\le b\le\delta_j)\wedge(a\ne b)\wedge T_i^a\wedge T_j^b.$$

Authorized licensed use limited to: Princeton University. Downloaded on August 19, 2009 at 15:33 from IEEE Xplore. Restrictions apply.

A. Test-Generation Algorithm

While targeting a nonvacuous matching for a given property, the testbench should not drive an instruction which causes a sequence of internal events that satisfies the property vacuously. Since each instruction h_i scheduled at time t causes internal events spanning from time t to time $t+\delta_i$, we must look ahead at least δ_i time steps to check whether the property becomes vacuously true if h_i is scheduled at time t. The following procedure performs this task.

Algo. 3.1: **Procedure *FindInstruction***(A, H)
// A *denotes the targeted assertion* and H*—the set of instructions*
// Let $\Delta = \max\{\delta_i, i = 1, N\}$
Step 1: Rewrite A in terms of the Boolean propositions over the next Δ cycles and temporal terms prefixed by Δ or more X operators
Step 2: Substitute the valuations caused by previous instructions in A to get L
Step 3: Let $\mathcal{P} \leftarrow \neg L \wedge \mathbf{Cons}$ (formally defined below)
Step 4: If $\mathcal{P}$ is satisfiable
$\widehat{h} \leftarrow$ An instruction at t from an assignment satisfying $\mathcal{P}$ else Print: No Solution and Abort
Step 5: return $\widehat{h}$

The term $\mathbf{Cons}$ of Step 3 expresses a few constraints about the system. First, we assume that only one instruction can be driven in each time step. This is expressed by the constraint

$$M : \text{mutex}\{h_i | i = 1, \ldots, |H|\}.$$

We also introduce the condition for each externally controllable internal event

$$C_j : \bigwedge_{k=0}^{\Delta} \left(\left(i_j^{t+k}\right) \leftrightarrow \bigvee_{x,y} \left(h_y^x\right) \right)$$

such that $t \le x \le (t+k)$ and $y = 1$ to $|H|$ and instruction h_y generates input event i_j at time $t+k$ when scheduled at time x. Intuitively

$$\left(i_j^{t+k}\right) \leftrightarrow \bigvee_{x,y} \left(h_y^x\right)$$

expresses the required condition for internal event i_j to occur at time $t+k$. The constraint $\mathbf{Cons}$ of Step 3 is defined as follows:

$$\mathbf{Cons} : \left(\bigwedge_{j=1}^{|\mathcal{I}|} C_j \right) \wedge M.$$

Theorem 1: FindInstruction never returns a vacuous instruction with respect to the targeted assertion A.

Proof: $(\mathbf{Cons} \Rightarrow L)$ becomes valid when $(\mathbf{Cons} \wedge \neg L)$ is unsatisfiable. Hence, we choose an instruction for which $(\mathbf{Cons} \wedge \neg L)$ remains satisfiable. □

IV. Model 3: Context-Driven Vacuity

Verification engineers advocate the use of auxiliary state machines (ASMs) in the development of formal property specifications [9]. The ASMs define the macrostates of the protocol between the DUT and the environment, and the formal assertions are typically local properties defined in the context of these macrostates of the ASMs.

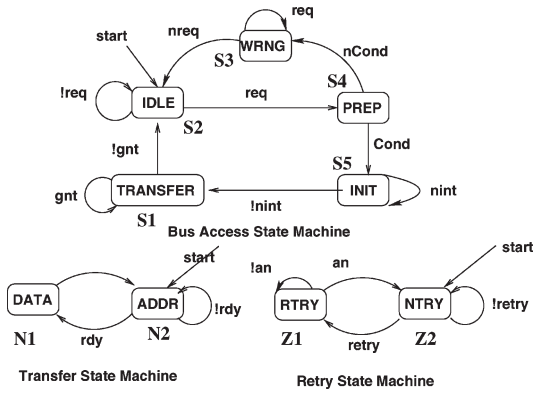

Fig. 1. ASMs.

We motivate the use of ASMs with a simple intuitive example and then explain the problem of generating tests to sensitize an assertion which has been written using the ASM states as its context. We shall refer to such assertions as context-triggered properties.

Example 2: Consider the specification of a master interface participating in a simple bus protocol with an arbiter and a slave device. Fig. 1 shows two ASMs for describing the macrostates of the protocol. $M1$ describes the state of the master during bus access in terms of req (request to arbiter) and gnt (grant from arbiter). $M2$ describes a more detailed state of the master during the transfer using the signal rdy (slave ready). Note that the ASMs involve a small fragment of the DUT interface signals. For our master interface, the following are other signals: 1) rw (indicating the nature of current transfer, write or a read); 2) validaddr (indicating the validity of the bus address); 3) abort (transfer terminated by the slave); and 4) delayed (delayed transfer).

The specification includes the following properties.

1) $\mathcal{P}_0$: If the transfer waits due to nonavailability of slave while the abort signal is low, then the delayed signal should be asserted in the next cycle.
2) $\mathcal{P}_1$: If the master is in the ADDR cycle in a write transfer, validaddr must be asserted in the next cycle.

Both properties are context-triggered. $\mathcal{P}_0$ applies only when the master is in the waiting phase, whereas $\mathcal{P}_1$ applies only when the master is in the ADDR cycle. It is easy to code these properties using the ASMs, as shown in the following:

$$\mathcal{P}_0 : G(\text{WAIT} \wedge \neg\text{abort} \Rightarrow X \text{ delayed})$$

$$\mathcal{P}_1 : G(\text{TRANSFER} \wedge \text{ADDR} \wedge \text{rw} \Rightarrow X\,\text{validaddr}).$$

Let us now consider the task of developing the same properties without using the ASMs. The main problem here is that the context states (such as WAIT, TRANSFER, and ADDR) cannot be characterized by Boolean propositions over the interface signals and require an encoding of the history of the protocol. For example, one might be tempted to express $\mathcal{P}_1$ as

$$G\left((\text{req} \wedge \text{gnt}) \wedge X(\text{req} \wedge \text{rdy} \wedge \text{rw}) \Rightarrow XX\,\text{validaddr}\right).$$

The antecedent part of the implication attempts to express the context TRANSFER ∧ ADDR. In this form, the antecedent appears nonintuitive. Moreover, the antecedent does not capture those scenarios where the DUT reaches the TRANSFER state after stuttering in one or more WAIT states. It is easy to see that a more accurate encoding of all possible history will make the antecedent almost unreadable and prone to coding errors. □

Authorized licensed use limited to: Princeton University. Downloaded on August 19, 2009 at 15:33 from IEEE Xplore. Restrictions apply.

The task of reaching a nonvacuous scenario for a context-triggered property consists of two parts. The first part is to guide simulation toward the states of the ASM which sensitize the context of the property. On reaching the context, the second part is similar to a DVG, except that we have to consider the constraints imposed by the transitions of the ASM also.

Example 3: Consider a context-triggered property defined over the states $S1$ and $S2$ of an ASM as given in the following:

$$G\left((S1 \wedge a \wedge X(S2)) \Rightarrow XX(b)\right).$$

The signal a is an input to the DUT, and the signal b is an output of the DUT. Let $C(S1, S2)$ denote the transition condition which enables the ASM to switch from $S1$ to $S2$.

To check this property nonvacuously, the simulation must first reach a state representing $S1$. On reaching $S1$, the testbench must drive the signal a, and also, it must enable the transition from $S1$ to $S2$ by choosing a valuation of the signals that satisfies $C(S1, S2)$. In other words, the testbench must choose a valuation of the signals it drives such that $a \wedge C(S1, S2)$ remains satisfiable. □

A. Test-Generation Algorithm

Given a context-triggered property L, we execute the following two games for reaching nonvacuous scenarios for L.

1) Game 1: Generate test vectors to cosimulate the ASM with the DUT and guide the simulation toward states satisfying the triggering context for L.
2) Game 2: Once the triggering context for L is reached, we play a DVG augmented by the constraints imposed by the ASM transitions.

The ASM states that satisfy the triggering context of L are extracted from L before Game 1 starts. Thereafter, Game 1 is played with the ASM as the reference. In each step of Game 1, the testbench asks the test generator for a valuation of the signals controlling the transitions between the ASM states. The following algorithm shows the steps of Game 1 for a given context-triggered property L.

Algo. 4.1: **Procedure *Game1Move***
InputVector $\widehat{I}$ **Game1Move**($\widehat{O}$, $\widehat{S}$, J, ASM)
// $\widehat{O}$ *denotes the present outputs of the DUT*
// $\widehat{S}$ *denotes the present ASM state of the DUT*
// Q *denotes the triggering context of* L
Step 1: If $\widehat{S}$ satisfies J, Return with success //*Context reached*
Step 2: Select a successor S' of $\widehat{S}$ in the ASM that is on some simple path to states satisfying Q.
Let $C(\widehat{S}, S')$ denote the enabling condition for $(\widehat{S}, S')$
Step 3: Substitute the values in $\widehat{O}$ in $C(\widehat{S}, S')$.
Step 4: Return any valuation of $\widehat{I}$ that does not refute $C(\widehat{S}, S')$

Once Game 1 returns with success, we start Game 2. The steps of Game 2 are shown in Algorithm 4.2. Let $C(S_i, S_j)$ denote the enabling condition for the transition from state S_i to state S_j in the ASM.

Algo. 4.2: **Procedure *Game2Move***
InputVector $\widehat{I}$ **Game2Move**($\widehat{O}$, $\widehat{S}$, L, ASM)
Step 1: Rewrite L in terms of the present-state Boolean propositions and X-guarded temporal properties to get L'
Step 2: Replace each term of the form $X(S')$ in L' with $C(\widehat{S}, S')$ to get $\widehat{L}$
Step 3: Execute Step 2 to Step 8 of Algo 2.1 with $\widehat{L}$.

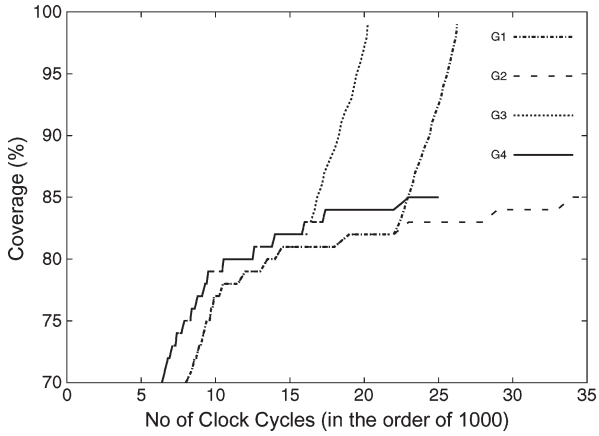

Fig. 2. Coverage progress in PLB and OPB (direct vacuity).

V. Experimental Results

We have applied the test methodologies on various test cases. For each case, we performed the verification in two phases. In Phase 1, we ran simulation using a standard constrained random testbench (CDR) for a large number of simulation cycles. At the end of this, we identified the set of uncovered properties. In Phase 2, we have used our methodologies targeting each of these properties. We show that Phase 2 achieves the verification closure much faster than continuing with Phase 1 for more number of cycles.

A. Results: Direct Vacuity

Table I shows the result of our tool for an industry standard assertion IP for the IBM CoreConnect BUS protocol [10]. It has two primary buses, namely, OPB and PLB (component details are given in Table I). In Phase 1, we have simulated the components with the CDR testbench for 21200 (for PLB) and 16150 simulation (clock) cycles (for OPB). The simulation takes 24 and 35 min for OPB and PLB, respectively. In Fig. 2, G2 and G4 show the coverage progresses of the CDR testbench for PLB and OPB. In both cases, a certain coverage is reached (nearly 70%) very quickly (within 6000 cycles for OPB and 8000 cycles for PLB). However, after that, the rate of increase in coverage became marginal. After Phase 1, 11 and 20 properties remained uncovered in OPB and PLB.

In Phase 2, we have used DVG as well as the CDR testbenches for the set of uncovered properties. In Fig. 2, G1 and G3 show the relative coverage progresses of the DVG testbench for PLB and OPB. The coverage improved dramatically when using our proposed (DVG) approach. Columns 11 and 12 of Table I show the required simulation cycles and time (in DVG) for covering the set of uncovered properties. The CDR testbench is also simulated for another 8850 cycles (13 min) for OPB and 13 800 cycles (22 min) for PLB in Phase 2 but with marginal growth in coverage.

B. Results: Context-Driven Vacuity

We have used the context-driven vacuity game (CDVG) on the same experimental setup explained in Section V-A. As before (see Section V-A), in Phase 1, the bus components have been simulated with the CDR testbench for 16 150 (for OPB) and 21 200 simulation (clock) cycles (for PLB). Similar to Fig. 2, in Fig. 3, G2 and G4 show the coverage progresses of the CDR testbench for PLB and OPB. After Phase 1, 11 and 20 properties remained uncovered in OPB and PLB.

Authorized licensed use limited to: Princeton University. Downloaded on August 19, 2009 at 15:33 from IEEE Xplore. Restrictions apply.

TABLE I
RESULTS ON IBM CORECONNECT

Device	# Inp.	# Outp.	# Prop.	Phase 1 (Cycles)	Phase 1 Time (Min)	Uncov. Prop.	CDR (Cycles)	Time (Min)	CDR Cov	DVG (Cycles)	Time (Min)	CDVG (Cycles)	Time (Min)
OPB interface													
master	6	6	24			6							
slave	5	4	11	16150	24	2	8850	13	84%	4000	6	3810	6
arbiter	5	4	11			3							
PLB interface													
master	10	12	33			10							
slave	8	10	24	21200	35	5	13800	22	85%	4480	7	4260	7
arbiter	6	11	23			5							

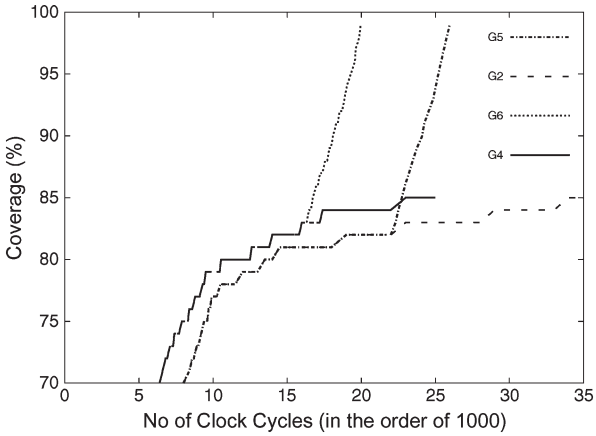

Fig. 3. Coverage progress in OPB and PLB (context-driven vacuity).

TABLE II
RESULTS ON DLX PIPELINE

	No. Prop	GenTest	RandS	% Impv
DHWDF	4	1200	3700	67.5
DHADF	4	1500	7000	78.5
STRH	2	750	2000	62.5
CNTRH	4	1800	7300	75.3
OOE	4	5000	22000	77.2

In Phase 2, we have used the CDVG scheme as well as the CDR testbench for covering the set of uncovered properties. In Fig. 3, G5 and G6 show the relative coverage progresses of the CDVG testbench for PLB and OPB. Columns 13 and 14 of Table I show the number of simulation cycles and time (using CDVG) for covering the set of uncovered properties.

C. Results: Indirect Vacuity

We have applied our methodology for validating properties on a five-stage DLX processor pipeline (based on models in [11]). We have tested common critical hazard conditions (e.g., data, structural (STRH) and control (CNTRH) hazards). For data hazards, we have considered both data-forwarding (DF) (DHWDF) and no DF (DHADF) schemes. We also considered out-of-order execution (OOE). VCS [7] is used as the simulator, and SystemVerilog [12] is used for testbench modeling. Column 2 (Table II) refers to the number of properties taken for the experiments. Columns 3 and 4 compare the number of simulation cycles required to cover these properties nonvacuously using our approach (GenTest) with that of coverage-driven random one (RandS). The last column shows the % improvement by our tool. For each property, we have considered at least five different nonvacuous instruction sequences. On complicated scenarios (e.g., OOE), our tool achieves a significant improvement on the simulation time (see Table II).

VI. CONCLUSION

Dynamic assertion-based verification remains widely used as model-checking techniques remain capacity-limited. Typically, constrained random simulation covers a lot of assertions in quick time but takes significant amount of time before reaching rare corner-case scenarios that are relevant to some critical assertions. In this paper, we have presented methods for guiding simulation toward such scenarios. These methods should be used at a time when the rate of gain in assertion coverage begins to decline.

The novelty of our approach is in treating the DUT as a black box and guiding simulation purely by external control. This helps in integrating our methods easily with existing simulation platforms.

REFERENCES

[1] A. Gupta, A. E. Casavant, P. Ashar, A. Mukaiyama, K. Wakabayashi, and X. G. S. Liu, "Property-specific testbench generation for guided simulation," in *Proc. ASP-DAC/VLSI Des.*, 2002, pp. 524–534.
[2] K. Heon-Mo and P. Mishra, "Functional test generation using property decompositions for validation of pipelined processors," in *Proc. Des. Autom. Test Eur.*, 2006, vol. 1, pp. 1–6.
[3] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *Proc. 7th ESEC/FSE*, 1999, pp. 146–162.
[4] J. A. Abraham, V. M. Vedula, and D. G. Saab, "Verifying properties using sequential ATPG," in *Proc. IEEE ITC*, 2002, pp. 194–202.
[5] M. Hsiao and J. Jain, "Practical use of sequential ATPG for model checking: Going the extra mile does pay off," in *Proc. 6th HLDVT Workshop*, 2001, pp. 39–44.
[6] O. Luethje, "A methodology for automated test generation for LISA processor models," in *Proc. 12th Workshop Synth. Syst. Integr. Mixed Inf. Technol.*, Kanazawa, Japan, Oct. 18/19, 2004, pp. 266–273.
[7] *VCS of Synopsys*. [Online]. Available: http://www.synopsys.com/products/simulation/simulation.html
[8] A. Banerjee, B. Pal, S. Das, A. Kumar, and P. Dasgupta, "Test generation games from formal specifications," in *Proc. DAC*, 2006, pp. 827–832.
[9] P. Dasgupta, *A Roadmap for Formal Property Verification*. New York: Springer-Verlag, 2006.
[10] *IBM CC Spec*. [Online]. Available: www-306.ibm.com/chis/techlib/techlib.nsf/techdocs/
[11] *DLX Spec*. [Online]. Available: http://www.opencores.org/pdownloads.cgi/list/aspida
[12] *SystemVerilog 3.1a Language Reference Manual*. [Online]. Available: www.eda.org/sv/SystemVerilog 3.1a.pdf

Authorized licensed use limited to: Princeton University. Downloaded on August 19, 2009 at 15:33 from IEEE Xplore. Restrictions apply.