Transaction Briefs

Automatic Verification Stimulus Generation for Interface Protocols Modeled With Non-Deterministic Extended FSM

Che-Hua Shih, Juinn-Dar Huang, and Jing-Yang Jou

***Abstract*—Verifying if an integrated component is compliant with certain interface protocol is a vital issue in component-based system-on-a-chip (SoC) designs. For simulation-based verification, generating massive constrained simulation stimuli is becoming crucial to achieve a high verification quality. To further improve the quality, stimulus biasing techniques are often used to guide the simulation to hit design corners. In this paper, we model the interface protocol with the non-deterministic extended finite-state machine (NEFSM), and then propose an automatic stimulus generation approach based on it. This approach is capable of providing numerous biasing strategies. Experiment results demonstrate the high controllability and efficiency of our stimulus generation scheme.**

***Index Terms*—Design automation, generators.**

I. Introduction

In the system-on-a-chip (SoC) era, designers tend to integrate a large number of components into a well-defined platform to accelerate the design process. To facilitate fast integration, components are usually compliant with certain interface protocol so that they can concordantly communicate with each other within the platform. Obviously, it becomes an important issue to preverify if a component can work correctly after being integrated into an SoC design. Many related studies [1]–[6] focus on building a monitor/checker to examine the behaviors of interface signals through dynamic simulation or formal techniques. The simulation-based approach is classical and widely used in most design environments. In this approach, a large amount of verification stimuli are demanded and applied to the design to hunt possible design errors. Nevertheless, the manual stimulus generation process tends to be time-consuming and error-prone. Hence, many research works [7]–[11] aim at automating the stimulus generation process.

To obtain massive stimuli automatically, it is intuitive to use a random stimulus generation approach. However, the major disadvantage is that the random process could produce invalid stimulus sequences violating the target specification. Instead, constraint-based verification methodology [12] is usually adopted to avoid this drawback. Constraints are actually formal specifications of design behaviors. A constraint-based stimulus generator produces only valid stimuli based on the given constraints. Previous research works typically use satisfiability (SAT) or binary decision diagram (BDD) solvers as their constraint-solving engines. The SAT method is a formal technique that is generally capable of solving a large number of complex constraints. A typical SAT solver often generates only one feasible solution under a given constraint set with no biasing capability. Alternatively, solving constraints with BDDs is relatively easier to be combined with the bit-level biasing approach. However, the weakness of the BDD-based solver is the memory explosion problem while handling a large set of constraints.

Yuan *et al.* [8] propose a general-purpose BDD-based stimulus generation approach. Constraints are represented as Boolean formulas with state variables. Then they conjoin related constraints into a single BDD before simulation. Next, BDDs are traversed in a top-down fashion to produce stimuli. Note that the bit-level biasing can be applied to adjust the branch probabilities of BDD nodes while performing the traversal. Another work by Shimizu *et al.* [9] targets on interface verification. First, authors write a list of interface constraints in a proprietary specification style. Next, they create BDDs with appropriate constraints on-the-fly instead of before simulation. In this way, BDDs can be smaller and thus solved more quickly. However, this approach needs to rebuild a new set of BDDs at every simulation cycle.

In this paper, we propose a method to develop an automatic verification stimulus generator (AVSG) for interface compliance verification. The major difference against previous approaches is that the proposed AVSG uses a non-deterministic extended finite state machine (NEFSM) as the interface protocol specification. As well, the diversified biasing strategies, including the transition-level, transaction-level, word-level, and bit-level biasing, provide users higher controllability over stimulus generation. Our AVSG is also capable of checking if the design under verification (DUV) conforms to the interface protocol during simulation. Furthermore, unlike SAT- or BDD-based constraint solvers, this generator can be easily implemented in synthesizable hardware description language (HDL). Therefore, it is feasible to dramatically speed up the verification process via a hardware accelerator or an emulator.

The rest of this paper is organized as follows. Section II introduces the NEFSM model. The stimulus generation, biasing methodology, and AVSG compiler are described in Section III. Section IV shows the experimental results. Finally, Section V concludes this paper.

II. Preliminaries

The extended finite-state machine (EFSM) model [13] is a finite-state machine extended with internal variables. It provides a more efficient way to describe the behavior of a sequential circuit and relaxes the state explosion problem suffered by traditional finite state machine models. Though the EFSM model has been widely used in many previous research works [14] and [15], we slightly modify the definition of the state transition to best fit our own need here.

Definition 1: For a variable V, its value set is D_V. For a finite set of variables $S = \{V_1, V_2, \ldots, V_n\}$, its value set D_S is the n-dimensional Cartesian product $D_{V_1} \times D_{V_2} \times \cdots \times D_{V_n}$.

Definition 2: An EFSM is a 7-tuple $(Q, \Sigma, \Delta, X, q_0, x_0, T)$:

Q	a finite set of states;
Σ	a set of inputs;
Δ	a set of outputs;
X	a set of variables;
q_0	the initial state, $q_0 \in Q$;
x_0	a set of initial values of variables in X;
T	a set of state transitions, each transition t is a 4-tuple (s_t, q_t, e_t, u_t), where;

Manuscript received October 05, 2007; revised March 03, 2008. First published March 10, 2009; current version published April 22, 2009.

The authors are with the Department of Electronics Engineering, National Chiao Tung University, Taiwan 300, R.O.C. (e-mail: matar@eda.ee.nctu.edu.tw; jdhuang@mail.nctu.edu.tw; jyjou@faculty.nctu.edu.tw).

Digital Object Identifier 10.1109/TVLSI.2008.2006042

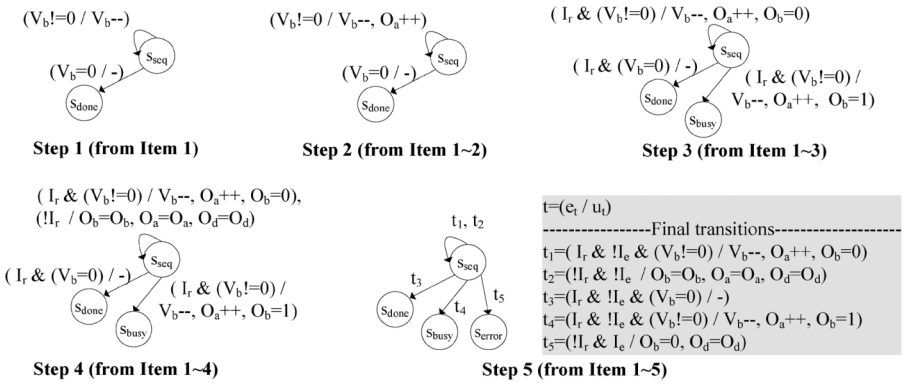

Fig. 1. NEFSM of the simplified AMBA AHB protocol.

s_t the current state, $s_t \in Q$;

q_t the next state, $q_t \in Q$;

e_t the transition enabling function, returning true(1)/false(0) to enable/disable the transition, $e_t : D_\Sigma \times D_\Delta \times D_X \rightarrow \{0, 1\}$;

u_t the update transformation function, updating the values of the subset $S \subseteq \Delta \cup X$, $u_t : D_\Sigma \times D_\Delta \times D_X \rightarrow D_S$.

Definition 3: Given an EFSM, for any state $s \in Q$ and its outgoing transition set $T_s = \{t | t = (s_t, q_t, e_t, u_t) \in T \text{ and } s_t = s\}$. If $\exists t_i$, $t_j \in T_s$, $t_i \neq t_j$ and $d \in D_\Sigma \times D_\Delta \times D_X$, such that both $e_{t_i}(d) = 1$ and $e_{t_j}(d) = 1$, then the given EFSM is an NEFSM. Otherwise, it is a *deterministic* EFSM.

In a deterministic EFSM, for a given combination of input, output, variable, and state values, there is at most one transition that can be enabled. On the contrary, an NEFSM allows multiple possible transitions. This non-determinism adequately formulates the common behaviors frequently appearing in most interface protocol specifications. For instance, a bus slave may choose to respond to a request immediately or to insert extra wait cycles before responding.

III. Methodology

A. Protocol Modeling

Since the interactions between the DUV and the rest of the system must obey a specific interface protocol, the first step of our approach is using an NEFSM to model the interface protocol with respect to the DUV. Due to the non-determinism of most existing interface protocol specifications, it is natural to model those protocols with NEFSMs. In order to clearly illustrate the modeling process, we introduce a protocol simplified from the AMBA AHB [16] as an example. It only considers two input signals (I_r and I_e) and three output signals (O_b, O_a, and O_d). The following is a part of the bus master specification about issuing a fixed-length incrementing burst to a slave.

Item 1) n-beat burst consists of n data transfers.

Item 2) Address (O_a) of the next transfer in a burst is equal to the address of the current transfer plus one.

Item 3) Slave asserts the ready signal (I_r) when it is ready for the current transfer. The master should send the values of the current data (O_d) and the next address. Besides, the master can choose to continue the burst or send a busy response $(O_b = 1)$ to temporarily suspend the next transfer.

Item 4) Slave can insert extra wait cycles by deasserting I_r. In this situation, all output signals of the master must hold their previous values.

Item 5) Slave can return an error response by asserting the error signal (I_e) and deasserting I_r simultaneously. In this situation, O_d must hold its previous value.

According to the previous specification, we build the corresponding NEFSM, shown in Fig. 1, step by step. The final NEFSM has four states (s_{seq}, s_{done}, s_{busy}, and s_{error}) and five transitions ($t_1 \sim t_5$) with an internal variable (V_b) for burst-length counting. Note that since we only consider a simplified partial specification here, some states have no outgoing transition in this case.

B. Stimulus Generation Flow

We develop a compiler that can automatically translate a given NEFSM model into the corresponding AVSG. The generated AVSG is capable of producing massive random stimuli fully compliant with the given interface protocol via the corresponding NEFSM.

A three-phase flow is proposed to automatically generate the stimulus on-the-fly based on the current DUV's response during simulation.

1) ***Evaluation***: At the current state, evaluate the enabling functions of all its outgoing transitions. The return values of the enabling functions are determined by the current values of the interface signals (Σ, Δ) and internal variables (X). A transition is put into the *next transition candidate set* (NTCS) only if its enabling function is evaluated true. Some transitions might be evaluated false and excluded from the NTCS. It actually means the current signal values on the interface prevent the AVSG from producing certain stimuli for the next cycle. In other words, the AVSG implicitly solves constraints presented by the given protocol during the stimulus generation process. Meanwhile, in case the NTCS is empty after the evaluation, it means the behavior of DUV must violate the protocol because it makes the AVSG find no valid move for the next cycle. That is, the AVSG can not only generate the valid stimuli but also serve as a protocol compliance checker at the same time.
2) ***Selection***: From the non-empty NTCS, randomly pick one as the next transition based on the given transition *weights*. A transition with a higher weight has a higher probability to be selected as the next transition. Hence, some sort of biasing strategies can be utilized here to meet different requirements from users. Meanwhile, the next transition becomes *determinate* at this phase no matter what selection strategy is in use.
3) ***Update***: Assign values to the outputs and variables according to the update transformation function of the selected transition. This phase consists of the constrained and unconstrained parts:
   a) *Constrained Part*: Outputs and variables explicitly constrained in the update transformation function must be assigned with the constrained values.
   b) *Unconstrained Part*: Outputs not constrained in the update transformation function can be randomly assigned with any valid values within their own domains. Again, some kind of biasing strategies can be used here.

TABLE I
BIASING INFORMATION I

Transaction-level Weight	Word-level Weight
$W_{t_1} = 80$	$W_{d=00} = 5$
$W_{t_2} = 40$	$W_{d=01} = 40$
$W_{t_3} = 40$	$W_{d=10} = 40$
$W_{t_4} = 20$	$W_{d=11} = 15$
$W_{t_5} = 100$	

Actually, the mission of the Update phase is to generate a complete and valid stimulus. After the Update phase, the NEFSM moves to the next state through the selected transition and then the stimulus generation process goes back to the Evaluation phase for the next cycle.

Now we use the AVSG built from the NEFSM in Fig. 1 to demonstrate this flow. Assume the current state is s_{seq}.

Case 1) ($I_r = 1, I_e = 0, O_a = 20, V_b = 4$).
The enabling functions e_{t_1} and e_{t_4} are both evaluated true at the Evaluation phase. It means the two corresponding transitions t_1 and t_4, are the next transition candidates. Next, at the Selection phase, one of the candidates would be chosen as the next transition. Assume the transition t_4 is selected, the constrained outputs and variables of u_{t_4} need to be updated as the defined values—assigning O_b, O_a, and V_b to 1, 21, and 3, respectively. The remaining unconstrained outputs can be assigned to any valid values. For instance, we can set O_d to 1. Finally, the current state moves from s_{seq} to s_{busy}.

Case 2) ($I_r = 1, I_e = 1$).
In this case, all enabling functions return false at the Evaluation phase. In other words, no possible valid transition exists and a protocol violation is detected by the AVSG. Meanwhile, it terminates the current simulation and reports the error.

C. Biasing

Biasing techniques can help verification engineers generate stimuli to hit desirable corner cases more easily. Our AVSG is capable of providing users to adjust various bias settings to guide the generated stimuli. This feature is extremely useful to exercise those uncovered scenarios to get a better simulation quality.

Transition-Level Biasing/Transaction-Level Biasing: Since each transition indicates certain interface behavior, we use the *transition-level biasing* to guide the state transition. Due to the non-determinism, there may exist multiple valid transitions after the Evaluation phase. A strategy is needed to pick one from these candidates. One method is to give each transition t an individual weight w_t. Then, the probability that a candidate transition t_i is selected is defined as

$$P_{t_i} = \frac{w_{t_i}}{\sum_{t_j \in NTCS} w_{t_j}}, \quad t_i \in \text{NTCS}.$$

For example, in Case 1) of the previous example, t_1 and t_4 are both transition candidates. If the biasing information is given as Table I, t_1 has a higher probability (80%) to be chosen than t_4 (20%) does. Furthermore, if preliminary simulation results do not exercise certain states or transitions, the related transition weights can be increased accordingly. Another similar approach is the *transaction-level biasing*. Users can define a meaningful transaction in terms of a sequence of transitions and then bias these transitions simultaneously.

Bit-Level Biasing/Word-Level Biasing: In typical protocols, many signals are defined in a group of bits, i.e., a word, instead of a single bit only. Therefore, the capability of the word-level biasing becomes critical and essential. In our approach, we allow the weight settings for different word values of d as shown in Table I to apply direct word-level biasing. This biasing setting simultaneously increases the appearance probabilities of "$d = 01$" and "$d = 10$". Note that, under bit-level biasing, the positive biasing of "$d = 01$" implies the negative biasing of "$d = 10$". Obviously, there is no way for bit-level biasing to achieve the distribution of word-level biasing given in Table I.

TABLE II
BIASING INFORMATION II

Biasing type	Weight
Word-level	$W_{b=0} = 3$
	$W_{b=1} = 1$
	Modified weight
Transition-level	$W'_{t_1} = 80 * (3/4) = 60$
	$W'_{t_2} = 40 * (4/4) = 40$
	$W'_{t_3} = 40 * (4/4) = 40$
	$W'_{t_4} = 20 * (1/4) = 5$
	$W'_{t_5} = 100 * (3/4) = 75$

In our approach, the word-level biasing settings can affect the generated stimuli in two manners. On the one hand, while the biased signal is in the unconstrained part of the Update phase, the signal's value can be directly produced via a weighted random number generator with the distribution specified by the word-level biasing. On the other hand, if the biased signal appears in the constrained part, i.e., the signal value relates to which transition is selected, the biasing effect could be reflected by changing the transition weights. Assume the word-level biasing list for an n-bit signal s is "$W_{s=0}, W_{s=1}, \ldots, W_{s=2^n-1}$" and $C^t_{s=i}$ is a binary variable indicating if "$s = i$" is feasible while selecting t as the next transition, for $0 \le i \le 2^n - 1$. While the original transition weight of the transition t is w_t, the modified transition weight according to the word-level biasing is

$$w'_t = w_t * \frac{\sum_{i=0}^{2^n-1} C^t_{s=i} * W_{s=i}}{\sum_{i=0}^{2^n-1} W_{s=i}}.$$

Table II shows an example of this mechanism. When the word-level weights of the signal b are given, the original transition weights, as shown in Table I, can be modified according to the update transformation functions of those transitions. While t_1 and t_4 constrain the value of b to 0 and 1, respectively, the modification ratio, $W'_{t_1}/W_{t1} : W'_{t_4}/W_{t4}$, is 3:1, which is the same as the word-level biasing setting of $b(W_{b=0} : W_{b=1})$. After this adjustment, the transitions tending to generate signal values with higher word-level weights should appear more frequently.

D. AVSG Compiler

We implement an AVSG compiler as shown in Fig. 2. The compiler first reads in the given NEFSM of the specific interface protocol and the biasing information. It then automatically produces a corresponding AVSG in target HDL format. The upper part in Fig. 2 is a typical simulation-based verification environment. DUVs are usually written in HDL. However, high-level test benches or stimulus generators are often developed in C/C++, so they need to interact with the HDL simulator via the Programming Language Interface (PLI). Extra simulation overhead is required due to the PLI communication need. Since our AVSG is implemented in native HDL, it can thus save simulation time compared to those approaches using PLI.

Fig. 3 shows the AVSG input/output (I/O) interface. The primary input ports (Σ) and output ports (Δ) of generated AVSG are defined in NEFSM model. Besides, it has an additional output port, "FAIL", to indicate whether any protocol violation occurs. This translation process can be completed in the following three steps.

Step 1) First, the compiler translates the enabling functions into HDL assignment statements. For example, the enabling function e_{t_1} in Fig. 1 can be translated into the following Verilog statement:

$$E_{t_1} = (I_r)\&\&(!I_e)\&\&(V_b != 0).$$

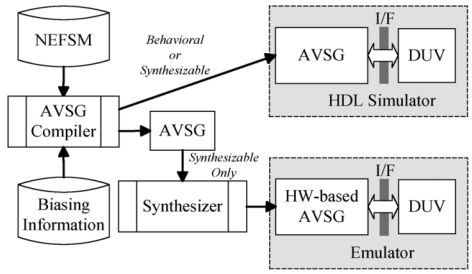

Fig. 2. AVSG generation flow.

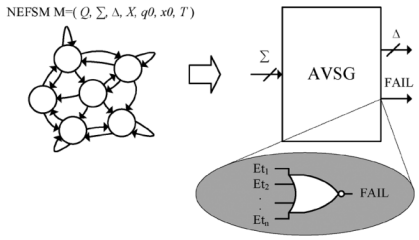

Fig. 3. AVSG I/O interface.

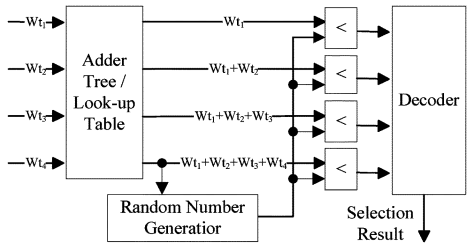

Fig. 4. Weighted selection procedure.

In this statement, E_{t_1} is a binary variable representing the evaluation result. The evaluation results for all outgoing transitions of the current state pass through an NOR gate and then drive the output signal "FAIL" as shown in Fig. 3.

Step 2) Unlike the deterministic machine, the NEFSM has a set of transition candidates. In our approach, the AVSG chooses the next transition according to the given transition weights (transition-level biasing). We use a weighted selection procedure as shown in Fig. 4 to realize this mechanism. First, it sums up the weights of transitions in NTCS via additions or a lookup table. Then, it generates a random number between 0 and the weight sum. According to this number and weight distribution, a decoder can determine the selection result.

Step 3) The major task of AVSG is to generate proper values for output signals which are constrained by the selected transition. On the one hand, the constraints are defined in the update transformation functions, and the compiler directly translates them into HDL-based assignment statements. For example, the update transformation function u_{t_1} can be implemented with three statements in Verilog

$$V_b = V_b - 1; \quad O_a = O_a + 1; \quad O_b = 0.$$

On the other hand, the weighted selection procedure is used here again to assign the unconstrained output signals with weighted random values (bit-level/word-level biasing).

Synthesizable AVSG: Practically, simulation jobs are very time-consuming for large complex SoC designs. Hence hardware accelerators or emulators are usually utilized to speed up the verification process around 100 $\sim$ 1000 times. The lower part of Fig. 2 demonstrates this acceleration environment. In general, SAT- or BDD-based generators are inherently hard to be synthesized down to hardware, and thus prevent such kind of acceleration. Conversely, the proposed compiler is capable of generating the AVSG in synthesizable HDL form on one condition—only synthesizable operators are allowed in the transition enabling and update transformation functions of the given NEFSM. In our experiences, the set of synthesizable operators are large enough for modeling most interface protocols. Meanwhile, the random number generator is required to implement the weighted selection schemes requested by the Selection and Update phases. This hardware implementation is similar to the hardware Lottery manager circuit in [17]. To make our AVSG fully synthesizable, a hardware-based linear feedback shift register (LFSR) [18] is used. The combination of these two efforts enables the truly hardware-based AVSG as well as the use of hardware acceleration techniques.

TABLE III
BASIC INFORMATION OF SELECTED DUVS

Design	Description	Protocol
AC97	Simple AC97 controller	WISHBONE
SPI	Serial Peripheral Interface	WISHBONE
PTC	PWM/Timer/Counter	WISHBONE
RGB2YCrCb	RGB-to-YCrCb Translator	AMBA AHB
CON	Convolution Calculator	AMBA AHB
MAC	Multiply-accumulator	AMBA AHB

IV. EXPERIMENTAL RESULTS

To demonstrate the effectiveness and efficiency of our approach, we use the WISHBONE [19] and AMBA AHB protocols as the test drivers. The experiments are conducted over a set of WISHBONE-compliant and AHB-compliant bus slave designs. The WISHBONE test cases are obtained from http://www.opencores.org/, and the others are internal designs. Table III shows the basic information of each design.

We first model both the WISHBONE and AHB master protocols in NEFSM. The resultant NEFSM of the WISHBONE requires 4 states and 17 transitions while AHB's requires 6 states and 46 transitions. Then the target AVSGs are directly generated from the corresponding NEFSMs through the proposed compiler.

The overall experimental results confirm that generated AVSGs are fully capable of providing a large amount of valid stimuli for all six designs. Next, we report certain detailed experimental results to show the power of biasing and the runtime efficiency of our AVSG.

Biasing: We apply the proposed biasing methods on the experiments over the design *CON*. We first show the effectiveness of the transaction-level biasing. Suppose each transition initially has equal weight and users want to exercise a 5-cycle transaction, "Nonseq $\rightarrow$ Busy $\rightarrow$ Seq $\rightarrow$ Busy $\rightarrow$ Seq". However, this transaction never occurs in the initial unbiased simulation of 1000 cycles. Then we increase the weights of related transitions (e.g., Nonseq$\rightarrow$Busy) by 10 times. Under the new bias setting, the expected transaction appears 19 times in first 1000 cycles. It demonstrates that the transaction-level biasing technique can help achieve higher functional coverage in shorter simulation cycles.

Another experiment focuses on performing the word-level biasing on the AHB signal *HBURST*. *HBURST* is a three-bit signal indicating the current burst type as shown in the first and second columns of Table IV. For certain verification need, we set the expected probabilities on different *HBURST* values in terms of the word-level weights. Note that no single bit-level bias setting on *HBURST* can produce the identical word-level biasing shown in the third column in Table IV. After 1 million simulation cycles, the appearance count of each burst type is reported in the last column. The results perfectly match the given bias setting. This experiment also implies that we can disable certain types

TABLE IV
RESULTS OF WORD-LEVEL BIASING

HBURST	Burst type	Weight	Count	
000	SINGLE	10	13022	(9.95%)
001	INCR	20	25996	(19.86%)
010	WRAP4	40	52377	(40.02%)
011	INCR4	5	6521	(4.98%)
100	WRAP8	15	19645	(15.01%)
101	INCR8	0	0	(0%)
110	WRAP16	0	0	(0%)
111	INCR16	10	13317	(10.18%)

TABLE V
RUNTIME ANALYSIS OF DIFFERENT STIMULUS GENERATORS

Design	Stimulus generator			
	PRSG	*AVSG*	SG_1	SG_2
AC97	117.56 s	123.04 s	136.92 s	139.81 s
SPI	19.45 s	24.03 s	36.60 s	37.92 s
PTC	14.88 s	17.42 s	31.52 s	33.90 s
RGB2YCrCb	11.10 s	12.20 s	23.97 s	24.46 s
CON	10.43 s	11.37 s	24.01 s	25.09 s
MAC	12.80 s	13.82 s	22.11 s	22.55 s
Ratio	0.92	1.00	1.36	1.41

of stimuli by setting the corresponding weights as zeros. This skill is extremely useful when a DUV does not fully implement all features specified in the interface protocol. In short, through the proposed biasing techniques, it becomes much easier to get the desired stimuli.

Performance Analysis: Shorter stimulus generation time should be always preferred during verification. To illustrate the performance of the AVSG, the runtime is compared with those of other stimulus generation methods. We build four simulation environments which contain different stimulus generators. The runtime (for 1 million simulation cycles) for six real designs in each simulation environment is reported in Table V. In the first environment, we use a pure random stimulus generator (PRSG) to produce stimuli. This is the most trivial way to generate massive random stimuli at virtually no cost. The second environment is to use our AVSG instead of the PRSG. From Table V, the difference of runtime required by the PRSG and our AVSG is quite small. It shows our AVSG can be as efficient as a PRSG. However, the PRSG generally produces invalid stimuli while the AVSG only generates stimuli fully compliant with the protocol. Besides, we build two BDD-based implementations, referred to as SG_1 and SG_2, which are similar to the stimulus generators in [8] and [9], respectively. Since PLI mechanism is required between the HDL simulator and BDD-solving engine for these two environments, the experimental results show that the two environments take averagely 36% and 41% more runtime than our approach.

Error Detection: While the NEFSM model describes the target interface protocol, the AVSG can also detect those design errors which violate the interface protocol. For demonstrating this feature, we do inject protocol-related errors into designs, and the experimental results show that the AVSG can indeed capture all these kinds of design errors. We also inject protocol-independent errors to correct designs. As expected, this kind of error can not be detected by our AVSG while the error effect does not affect the interface behavior. To detect those internal design bugs, users need to build other checkers to investigate the simulation results and the AVSG purely serves as a stimulus generator in this case.

Hardware Synthesis: As mentioned, our AVSG can be mapped to real hardware for emulation through a logic synthesizer. The synthesizer reports that only 1.8 and 3.7 K gates are required to implement the AVSGs of the WISHBONE and AHB protocols, respectively. This result clearly shows that the proposed AVSG can be easily and cost-effectively integrated into a emulator-based verification flow.

V. CONCLUSION

In this paper, we propose a constrained random stimulus generation approach for interface protocol verification. In this approach, the NEFSM model is used to represent the interface specification and then automatically translated into a dedicated AVSG. This AVSG can produce a large amount of valid random stimuli and simultaneously check the correctness of the interface behavior. In addition, it can be implemented in synthesizable HDL to enable the hardware acceleration. By supporting many biasing methods, the AVSG provides users much better controllability over where a simulation run heads for. The experimental results demonstrate that these biasing methods do successfully generate appropriate stimuli as expected. Moreover, the extremely low overhead in simulation time shows the high efficiency of our AVSG. Hence, this approach can indeed improve the simulation quality as well as speed up the verification process.

REFERENCES

[1] K. Shimizu, D. L. Dill, and A. J. Hu, "Monitor-based formal specification of PCI," in *Proc. Int. Conf. Formal Methods Comput.-Aided Des.*, Nov. 2000, pp. 335–353.
[2] A. Nightingale and J. Goodenough, "Testing for AMBA compliance," in *Proc. IEEE Int. ASIC/SOC Conf.*, Sep. 2001, pp. 301–305.
[3] M. T. Oliviera and A. J. Hu, "High level specification and automatic generation of IP interface monitors," in *Proc. Des. Autom. Conf.*, Jun. 2002, pp. 129–134.
[4] A. J. Hu, J. Casus, and J. Yang, "Efficient generation of monitor circuits for GSTE assertion graphs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Nov. 2003, pp. 154–159.
[5] A. Roychoudhury, T. Mitra, and S. R. Karri, "Using formal techniques to debug the AMBA system-on-chip bus protocol," in *Proc. Des., Autom. Test Europe Conf. Exhibition*, Mar. 2003, pp. 828–833.
[6] H.-M. Lin, C.-C. Yen, C.-H. Shih, and J.-Y. Jou, "On compliance test of on-chip bus for SOC," in *Proc. Asia South Pacific Des. Autom. Conf.*, Jan. 2004, pp. 328–333.
[7] K. Ara and K. Suzuki, "A proposal for transaction-level verification with component wrapper language," in *Proc. Des., Autom. Test Europe Conf. Exhibition*, Mar. 2003, pp. 82–87.
[8] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, "Modeling design constraints and biasing in simulation using BDDs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Nov. 1999, pp. 584–589.
[9] K. Shimizu and D. L. Dill, "Deriving a simulation input generator and a coverage metric from a formal specification," in *Proc. Des. Autom. Conf.*, Jun. 2002, pp. 801–806.
[10] J. Yuan, C. Pixley, A. Aziz, and K. Albin, "A framework for constrained functional verification," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Nov. 2003, pp. 142–145.
[11] M. A. Iyer, "RACE: A word-level ATPG-based constraints solver system for smart random simulation," in *Proc. Int. Test Conf.*, Sep. 2003, pp. 299–308.
[12] J. Yuan, C. Pixley, and A. Aziz, *Constraint-Based Verification*. New York: Springer Science Business Media, Inc., 2006.
[13] G. V. Bochmann and J. Gecsei, "A unified method for the specification and verification of protocols," in *Proc. Int. Federation for Inf. Process. Congr.*, Aug. 1977, pp. 229–234.
[14] K.-T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *Proc. Des. Autom. Conf.*, Jun. 1993, pp. 86–91.
[15] D. Lee and M. Yannakakis, "Optimization problems from feature testing of communication protocols," in *Proc. Int. Conf. Netw. Protocols*, Oct. 1996, pp. 66–75.
[16] ARM Limited, Cambridge, U.K., "AMBA Specification (Rev. 2.0)," May 1999.
[17] K. Lahiri, A. Raghunathan, and G. Lakshminarayana, "Lotterybus: A new high-performance communication architecture for system-on-chip design," in *Proc. Des. Autom. Conf.*, Jun. 2001, pp. 15–20.
[18] P. H. Bardell, W. H. McAnney, and J. Savir, *Built-in Test for VLSI: Pseudorandom Techniques*. New York: Wiley, 1987.
[19] OpenCores Organization, "Specification for the: WISHBONE SOC interconnection architecture for portable IP cores, Rev. B.3," Sep. 2002. [Online]. Available: http://www.opencores.org/