FSM-Based Transaction-Level Functional Coverage for Interface Compliance Verification

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

Department of Electronics Engineering
National Chiao Tung University
Hsinchu, Taiwan, R.O.C.
e-mail:{powmei, matar}@eda.ee.nctu.edu.tw, jdhuang@mail.nctu.edu.tw, jyjou@faculty.nctu.edu.tw

Abstract – Interface compliance verification plays a very important role in modern SoC designs. In order to perform a quantitative analysis of simulation completeness, adequate coverage metrics are mandatory. In this paper, we propose a finite state machine (FSM) based transaction-level functional coverage methodology for interface compliance verification. A language, State-Oriented Language (SOL), is developed to specify functional transactions mainly at the higher FSM level instead of lower logic or signal level. By utilizing SOL, it is simple and rigorous to specify interesting transactions from the specification FSM of the target interface protocol. Experimental results show that the proposed methodology can effectively improve the verification quality as well as increase the efficiency of regression verification.

1. Introduction

In designing a modern system-on-a-chip (SoC), the platform-based design methodology with reusable intellectual property (IP) cores is usually adopted to accelerate the design and verification process [1]. Each pre-verified IP core is wrapped with certain interface logic and integrated into a system platform which is based on that interface protocol. In order to ensure that each component can concordantly communicate with others within the system, it is very important to guarantee that the interface logic of each utilized IP core conforms to the protocol. Hence, interface compliance verification becomes an essential part of the SoC verification flow.

Though there are numerous existing functional verification methods, simulation is still the most commonly used technique. During simulation, coverage metrics are usually adopted to perform a quantitative analysis of simulation completeness. Coverage metrics can not only measure how well a design is verified objectively but also help improve the quality of verification patterns. That is, they are capable of guiding either direct (deterministic) or random patterns to target those unverified design corners. Therefore, exploring adequate coverage metrics is a very crucial issue in today's functional verification.

In general, there are two major categories of coverage metrics [2]: code coverage and functional coverage. Code coverage methods concentrate on identifying which part of the hardware description language (HDL) code has been executed in the design under verification (DUV). That is, they measure how much of the HDL implementation has been exercised [3-6]. For example, statement coverage, branch coverage, and condition coverage are well-known code coverage metrics. However, the fundamental issue of all code coverage metrics is that they can only measure how well the structural HDL code has been exercised. They are not sufficient to represent the whole functionality of the design specification. Namely, the verification quality is generally considered not enough for modern complex SoC designs even if a high code coverage is achieved. Thus, the functional coverage is usually applied to further boost the verification quality.

Functional coverage, as its name implies, focuses on the design functionality. It measures how much of the original design specification has been verified. That is, the coverage is independent of the details of HDL implementation, and thus is considerably hard to measure. Many methods are proposed to address this issue. In [7], a user-defined cross-product coverage measurement tool is developed. In [8-9], the cross-product functional coverage is further improved either in quality or efficiency. In [10-11], the specification must be first given as a proprietary graph. Then the functional coverage analyzer can be automatically generated by traversing the graph. The methods mentioned above do really help interface compliance verification. However, these techniques generally require users to specify what they want to cover in proprietary input formats or languages uncommon to typical designers.

In this paper, we propose a transaction-level functional coverage methodology and provide a means to specify functional transactions at a higher FSM level, which is popular and familiar to most designers. First, the interface protocol is given as a specification FSM (spec FSM) by using the concepts in [12-13]. Then *a transaction can be defined as a specific sequence of state transitions within the spec FSM*. Meanwhile, we develop a transaction description language, State-Oriented Language (SOL), which is capable of modeling diverse state transition sequences precisely and rigorously. The transactions can then be specified in an easier and more readable way even by common designers. Moreover, the specified transactions with the spec FSM can be further translated into the corresponding functional coverage analyzer automatically.

The rest of this paper is organized as follows. Section 2 introduces the basic concepts and the related works of transaction-level functional coverage. In Section 3, the state-based transaction description language SOL and the details of our verification methodology are presented. Section 4 demonstrates the proposed methodology with the AMBA AHB slave protocol and shows the experimental results. Finally, the conclusions are given in Section 5.

2. Transaction-level functional coverage

As mentioned, functional coverage is favorable to improve the verification quality. Transaction-level functional coverage is one of the commonly used methods to measure the functional coverage for an interface design [13-16]. An interface specification usually defines a set of different transaction types. A transaction can be considered as the transfer of data and control over an interface to perform certain basic operation. For example, a transaction can be a 4-beat burst or an 8-beat burst, or a 4-beat burst followed by an 8-beat one. Transaction-level functional coverage is generally measured by how many types of transactions are exercised. However, two designs may have different sets of interesting transactions even if they comply with the same interface protocol. Therefore, the interesting transactions of a given design are usually derived manually.

Several approaches are proposed for the transaction-level functional coverage. For M-path coverage [13], the protocol is first modeled as a spec FSM. Then an M-path is defined as a path of state transitions which can form a complete bus transfer in the FSM model. In other words, an M-path, which is a finite sequence of state transitions, is actually a simple transaction. M-paths are used as the targets for coverage measurement.

In [14], Component Wrapper Language (CWL) is used to describe signal sequences based on regular expressions. In CWL, the input and output signals must be declared first. Then signal values at each cycle are defined as signal sets. Next, each simple transaction is modeled by utilizing the defined signal sets. Finally, a more complex transaction can be built up by assembling simple ones. In this approach, values of individual signals are required when describing thorough transactions. If the interesting transactions are getting more complex, it might be troublesome and time-consuming to author the corresponding CWL descriptions.

In general, it is tedious and error-prone for human to specify transactions if the detailed signal values are required. To cope with this issue, it is a better idea to provide a simple, human-friendly, rigorous, and systematic way to specify transactions at a higher level of abstraction instead of at the signal level. In our work, the interface protocol is specified as a spec FSM by using the methods in [12-13]. *A transaction can then be defined as a specific sequence of state transitions. This enables the use of states in the spec FSM as basic elements to describe transactions.* The proposed method can raise the transaction description to the FSM level which is well understood by most designers. It facilitates the encapsulation of the details of low-level signals so that the detailed signal values at each cycle are no longer required. Hence, one can put more emphasis on the functionality at the familiar FSM level.

3. Proposed approach

3.1. Our methodology

In this paper, we propose an FSM-based transaction-level functional coverage methodology. In order to provide a means to specify transactions at the FSM level, we develop a transaction description language, State-Oriented Language (SOL), mainly based on the Property Specification Language (PSL) [17]. Because PSL provides a richer set of expressive and readable language constructs than typical regular-expression-based approaches do, SOL adopts most PSL constructs used to describe temporal sequences. In SOL, the PSL-like syntax is used to represent *a sequence of state transitions.* Though SOL is similar to PSL, the fundamental conceptual difference between them is that *SOL uses states as the atomic elements when defining a transaction.* Hence, it is easier for designers to author complex state-based transactions by using SOL.

The flow of our methodology is illustrated in Figure 1. The interface protocol needs to be specified as a spec FSM first. Note that the spec FSM can be translated into an interface protocol checker [13]. Meanwhile, the interesting transactions are manually specified by using SOL. These transactions with the spec FSM are further translated into a functional coverage analyzer automatically. Next, we simulate the whole system, including the DUV, verification patterns, checker, and coverage analyzer. According to the outcome of the checker, we can know if the DUV conforms to the interface protocol. From the coverage analyzer, the report tells how many interesting transactions have been verified or not. Moreover, the coverage information can guide the development of either direct or random patterns to hit those unverified corner cases.

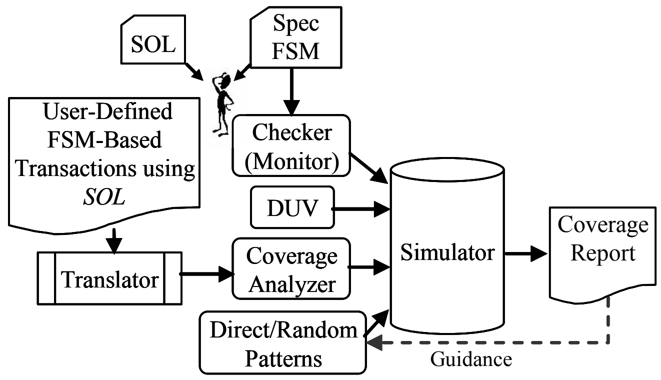

Figure 1. The flow of our methodology.

3.2. The transaction description language SOL

The syntax of SOL is based on the following principles:

- Since a transaction is defined as a specific sequence of state transitions in the spec FSM. *States* are used as basic elements to describe sequences.
- In order to keep the spec FSM as simple as possible, extra signals can be included in additional to the states while defining a transaction.
- A sequence can be defined once as a *named sequence* and then be reused later. The *assignment* operator is used to define a named sequence. The left-hand-side of the assignment operator becomes a synonym for the sequence on the right-hand-side.
- Sequence name is enclosed in *braces* when referred.
- A *sequence set* comprises one or more sequences. Sequences are enclosed in *angle brackets* and separated by *commas*.

The syntax of SOL is briefly introduced below (shown in shaded area). The FSM shown in Figure 2 is taken as an example to introduce operators in SOL.

3.2.1. Extra signal qualification (“”). Extra signals can be qualified while making a state transition. The *Boolean expression* built from the extra signals should be enclosed in *double quotes*.

3.2.2. Concatenation (;). Two sequences can be concatenated into one by the concatenation operator.

Example 1 In Figure 2(a), T1 is a transaction with the state transitions that starts from S1, then moves through S3, S4, and ends at S1.

T1: S1 → S3 → S4 → S1

T1 = { S1 ; S3 ; S4 ; S1 };

Example 2 In Figure 2(b), T2 is another transaction with the same state transitions sequence as T1 while the extra signal V must be true when moving from S1 to S3.

T2 : S1 $\xrightarrow{V==1}$ S3 → S4 → S1

T2 = { S1 “V == 1” ; S3 ; S4 ; S1 };

3.2.3. Repetition ([ ]). The repetition operators are used to describe repeated concatenations of a sequence. There are three types of the repetition operators: consecutive repetition ([*]), non-consecutive repetition ([=]), and goto repetition ([→]).

(a) consecutive repetition ([*]).

Example 3 In Figure 2(a), T3 is a transaction with the state transitions that starts from S1, moves to S2, and stays at S2 for three consecutive cycles, then ends at S1.

T3 : S1 → S2 → S2 → S2 → S1

T3= { S1 ; S2[*3] ; S1 };

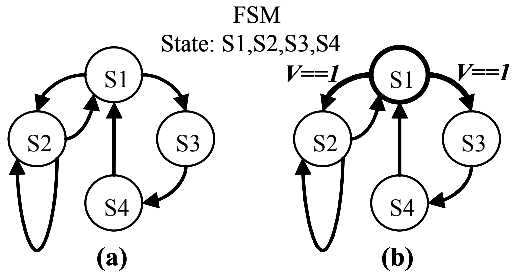

Figure 2. An example FSM.

Example 4 In Figure 2(a), T4 is a transaction with the state transitions that starts from S1, moves to S2, and stays at S2 for one to five consecutive cycles, then ends at S1.

T4 : S1 → S2 (1~5 cycles) → S1

T4 = { S1 ; S2[*1:5] ; S1};

(b) non-consecutive repetition ([=]).

Example 5 In Figure 2(a), T5 is a transaction with the state transitions that starts from S1, and then visits S2 three times. The visits of S2 need not to be in consecutive cycles. In addition, T5 holds after the 3rd S2 is visited and still holds before the 4th S2 appears.

T5 : S1 →…→ S2 →…→ S2 →…→ S2 →…→ S2 →…

T5 = { S1 ; S2[=3] };

(c) goto repetition ([→]).

Example 6 In Figure 2(a), similar to T5, T6 is also a transaction with the state transitions that starts from S1, and then moves to S2 three times (can be non-consecutive). In addition, T6 holds only at the cycle in which the 3rd S2 is visited.

T6 : S1 →…→ S2 →…→ S2 →…→ S2 →…→ S2 →…

T6 = { S1 ; S2[→3] };

3.2.4. Sequence AND (&&). The transaction comprising two sequences using the sequence AND operator holds only if both sequences hold and complete at the same cycle.

Example 7 In Figure 2(a), similar to T6, T7 is also a transaction with the state transitions that starts from S1, and then visits S2 three times (can be non-consecutive). However, S3 is strictly not allowed showing up in the sequence T7.

T7 : S1 →…(!S3) → S2 →…(!S3) → S2 →…(!S3) → S2

T7 = { S1 ; {S3[=0]} && S2[→3] };

3.2.5. Sequence OR (|). The transaction comprising two sequences using the sequence OR operator holds if one of two alternative sequences holds.

Example 8 In Figure 2(a), T8 is a transaction shown below,

T8 : S1→S3→S4→S1 OR S1→S2→S2→S2→S1

T8 = { {S1;S3;S4;S1} | {S1;S2[*3];S1} };

Note that above two sequences are previously defined as T1 and T3. Hence, T8 can also be defined in terms of these named sequences.

T8 = { {T1} | {T3} };

3.2.6. Sequence fusion (:). Similar to the concatenation operator, a sequence fusion operator concatenates two sequences overlapping by one cycle.

<u>Example 9</u> In Figure 2(a), T9 is a transaction shown below,

T9 : S1→S3→S4→S1→S2→S2→S2→S1

T9 = { S1;S3;S4;S1;S2[*3];S1 };

T9 can also be treated as two sequences that overlap each other for one cycle as shown below:

T9 : S1→S3→S4→S1 : S1→S2→S2→S2→S1

T9 = { {S1;S3;S4;S1} : {S1;S2[*3];S1} };

Again, T9 can also be defined in terms of T1 and T3.

T9 = { {T1} : {T3} };

3.2.7. Sequence set cross (**). A sequence set cross operator is used to represent a set of back-to-back consecutive transactions.

<u>Example 10</u> Assume the following 8 transactions are interesting.

{{T1}:{T3}:{T8}}; {{T1}:{T4}:{T8}}; {{T1}:{T3}:{T9}}; {{T1}:{T4}:{T9}}; {{T2}:{T3}:{T8}}; {{T2}:{T4}:{T8}}; {{T2}:{T3}:{T9}}; {{T2}:{T4}:{T9}};

The following expression utilizing the sequence set cross operator provides a much more elegant but equivalent representation for the set of 8 interesting transactions.

<{T1},{T2}> ** <{T3},{T4}> ** <{T8},{T9}>;

3.3. SOL examples

To apply our methodology, the interface protocol should be given as a spec FSM first. The details about how to construct a spec FSM can be found in [12-13]. The AMBA AHB slave interface protocol [18] is adopted here to demonstrate how to define transactions in SOL. The spec FSM of the simplified AMBA AHB slave protocol is given in Figure 3.

<u>Example 1</u> 1-beat burst transaction.

A 1-beat burst transaction basically means the given design moves to the state NSEQ/SEQ (S1) one time and can not move to the state ERROR (S4), i.e.,

{{S4[=0]} && {S1[→1]}}

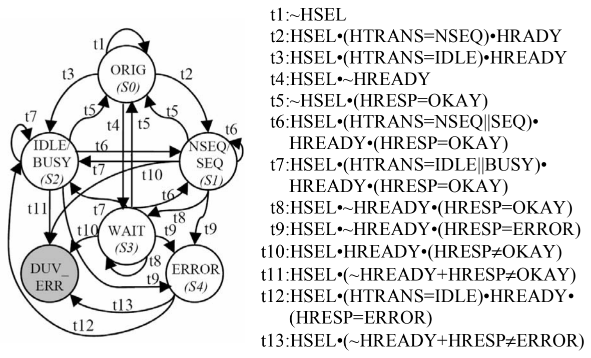
ORIG (S0)
IDLE/ BUSY (S2)
NSEQ/ SEQ (S1)
WAIT (S3)
ERROR (S4)
DUV_ ERR
t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13
t1:~HSEL
t2:HSEL•(HTRANS=NSEQ)•HRADY
t3:HSEL•(HTRANS=IDLE)•HREADY
t4:HSEL•~HREADY
t5:~HSEL•(HRESP=OKAY)
t6:HSEL•(HTRANS=NSEQ||SEQ)• HREADY•(HRESP=OKAY)
t7:HSEL•(HTRANS=IDLE||BUSY)• HREADY•(HRESP=OKAY)
t8:HSEL•~HREADY•(HRESP=OKAY)
t9:HSEL•~HREADY•(HRESP=ERROR)
t10:HSEL•HREADY•(HRESP≠OKAY)
t11:HSEL•(~HREADY+HRESP≠OKAY)
t12:HSEL•(HTRANS=IDLE)•HREADY• (HRESP=ERROR)
t13:HSEL•(~HREADY+HRESP≠ERROR)

Figure 3. The spec FSM of the simplified AMBA AHB slave protocol.

In addition, a 1-beat burst transaction consists of two cases. One starts from the state ORIG (S0), which indicates the slave is just selected and going to do the first transaction. The other starts from the state NSEQ/SEQ (S1), which implies the slave is already selected and going to do another transaction. Besides, the signal HBURST must be set to 0 for a 1-beat burst transaction.

(1) starting from the state ORIG (S0) :

One_S0 = {S0 "*HBURST==0*";{S4[=0]}&&{S1[→*1*]}};

(2) starting from the state NSEQ/SEQ (S1) :

One_S1 = {S1 "*HBURST==0*";{S4[=0]}&&{S1[→*1*]}};

The 1-beat burst transaction is composed of the sequence One_S0 and the sequence One_S1 by using a sequence OR operator. That is,

One = {{One_S0} | {One_S1}};

<u>Example 2</u> 4-beat burst transaction.

Similar to a 1-beat burst transaction, a 4-beat burst one also consists of two cases. But the design must visit the state NSEQ/SEQ (S1) four times. The signal HBURST should also be set to 2 or 3 for a 4-beat transfer.

(1) starting from the state ORIG (S0) :

Four_S0 = {S0"*HBURST==2* || *HBURST==3*";
{S4[=0]} *&&* {S1[→*4*]}};

(2) starting from the state NSEQ/SEQ (S1) :

Four_S1 = {S1 "*HBURST==2* || *HBURST==3*";
{S4[=0]} *&&* {S1[→*4*]}};

The 4-beat burst transaction can then be written as,

Four = {{Four_S0} | {Four_S1}};

<u>Example 3</u> A 4-beat burst transaction instantly followed by an 8-beat write burst transaction.

A 4-beat burst transaction (i.e., Four) is defined before, and an 8-beat write burst transaction (i.e., EightWrite) can also be specified in the similar way. Since the required transaction can be defined by fusing these two transactions, it can be written as {{Four}:{EightWrite}}; .

4. Experiments

4.1. Experimental environment

To demonstrate our methodology, we choose the AMBA AHB slave interface protocol [18] as an example. The spec FSM of the simplified AHB slave protocol is given in Figure 3. Figure 4 illustrates the experimental environment used in this work. It consists of three parts: a DUV, a constraint-driven random pattern generator, and the proposed verification framework.

(1) The experiments are conducted over three real AHB slave designs. The basic information of these designs is shown in Table 1. The design RGB2YCrCB is an RGB-to-YCrCB color space converter. The design MAC is a multiply-accumulator. The design Convolution is a convolution calculator to be used in discrete wavelet transfer.

(2) The constraint-driven random pattern generator is an AHB master which generates verification patterns based on

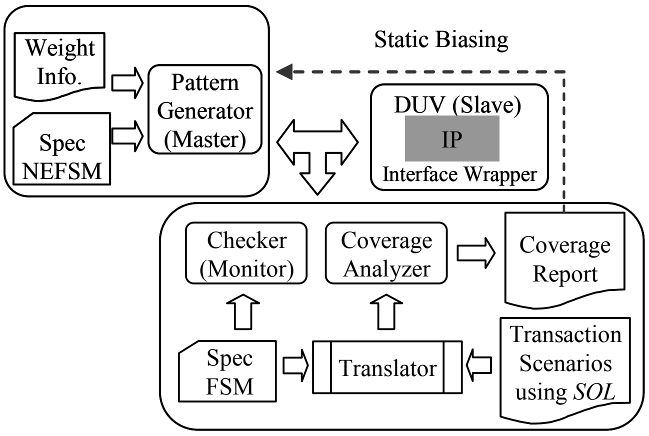

Figure 4. Experimental environment.

an NEFSM (Non-deterministic Extended FSM) with the weighted state transitions. The weight of each transition is configurable. The transitions are assigned with an *equal weight* initially.

(3) We develop a translator which accepts the spec FSM and user-defined SOL transactions then produces the corresponding coverage analyzer. The reported coverage is used to help statically bias the random pattern generator to create more effective verification patterns.

4.2. Experimental results

Two experiments are conducted: coverage comparison and efficiency improvement. In the first experiment, four coverage results (state, state transition, M-path, and our transaction coverage) are compared for three designs, respectively. In the second experiment, the coverage information is sent back to bias the random pattern generator to produce more effective patterns.

4.2.1. Coverage comparison

Case 1. The interesting transactions are defined as 10 basic read and write transactions, e.g., {OneRead};, {OneWrite};, {FourRead};, etc.

The comparison results are shown in Table 2. For the design RGB2YCrCb, it takes 4/16/82/492 cycles to reach 100% state/transition/M-path/transaction coverage. As the state/transition/M-path coverage reach 100%, the transaction coverage is only 0/10/20%. For the other two designs, the results are similar. It is observed that the transaction coverage is very low while the other three coverage metrics reach 100%.

Table 1. Design information.

Design	Supported AHB responses	# of state/transition/M-path
RGB2YCrCb	OKAY	3 / 8 / 14
MAC	OKAY, ERROR	4 / 10 / 12
Convolution	OKAY (wait)	4 / 10 / 16

Table 2. Coverage comparison for Case 1.

Design	Coverage	# of cycles to reach 100%	Transaction coverage (%)
RGB2YCrCb	State	4	0 (0/10)
	Transition	16	10 (1/10)
	M-path	82	20 (2/10)
	Transaction	492	100 (10/10)
Design	Coverage	# of cycles to reach 100%	Transaction coverage (%)
MAC	State	61	30 (3/10)
	Transition	61	30 (3/10)
	M-path	33	10 (1/10)
	Transaction	9644	100 (10/10)
Design	Coverage	# of cycles to reach 100%	Transaction coverage (%)
Convolution	State	12	10 (1/10)
	Transition	47	20 (2/10)
	M-path	102	30 (3/10)
	Transaction	787	100 (10/10)

Case 2. Make the interesting transactions more complex by adding 15 more transactions with BUSY/WAIT (e.g., {OneWithWAIT};,{FourWithBUSY};,etc.) and 25 consecutive transactions (e.g., <{Incr},{One},{Four},{Eight},{Sixteen}>** <{Incr},{One},{Four},{Eight},{Sixteen}>;).

The comparison results are shown in Table 3. For the design Convolution, it still takes 12/47/102 cycles to reach 100% state/transition/M-path coverage. But it takes 11135 cycles to reach 100% transaction coverage. As the state/transition/M-path coverage reach 100%, the transaction coverage is only 4/8/12%. It is shown that the transaction coverage is even lower than that in **Case 1** as the other three coverage metrics reach 100%.

We get some conclusions from the above 2 cases. While the set of interesting transactions becomes larger and more complex, it needs a significantly (non-linearly) longer simulation time to reach 100% transaction coverage. Moreover, even the state/transition/M-path coverage reach 100%, the transaction coverage can still be extremely low. The situation is getting worse when more complicated transactions are concerned. It means that even a pattern set developed to reach 100% state/transition/M-path coverage may not provide a satisfied functional coverage. Experimental results exactly show that the classical coverage metrics are not capable of providing enough verification quality.

4.2.2. Efficiency improvement

After analyzing the coverage report of **4.2.1 Case 2**, we find the major reason why so many cycles are required to reach 100% transaction coverage is the seldom occurrence of BUSY transactions. Hence, it is possible to reduce the simulation time by statically biasing the pattern generator. The biasing information is shown in Table 4.

In *bias1*, we intuitively increase the weights of transitions that can generate BUSY transactions. This biasing indeed decreases the simulation time to 1864 cycles, which is only 16.7% of the original one. In *bias2*, the weights of INCR burst, 1-beat burst, 4-beat burst, 8-beat

Table 3. Coverage comparison for Case 2.

Design	Coverage	# of cycles to reach 100%	Transaction coverage (%)
Convolution	State	12	4 (2/50)
	Transition	47	8 (4/50)
	M-path	102	12 (6/50)
	Transaction	11135	100 (50/50)

Table 4. Efficiency improvement.

Design	Bias	# of cycles to reach 100%	Factor
Convolution	*equal weight*	11135	1
	bias1	1864	0.167
	bias1 + bias2	981	0.088

burst, and 16-beat burst are given in *decreasing order* because the BUSY transaction takes place more frequently in long-beat transfers. Combining *bias1* with *bias2*, the simulation time can be further decreased to 981 cycles, which is only 8.8% of the original one.

The results show that the coverage information can help bias the random pattern generator to create more effective patterns and help verify the DUV in a shorter time. This technique is extremely useful while developing a regression verification environment in which the compact and effective pattern suites are crucial to minimize the required simulation time. That is, the proposed methodology can increase the efficiency of the regression verification process.

5. Conclusions

In the paper, we propose an FSM-based transaction-level functional coverage methodology for interface compliance verification. To provide a familiar, user-friendly, but still rigorous, and systematic way to specify transactions at a higher FSM level, we develop a PSL-like transaction description language SOL. The expressive power of SOL is generally stronger than that of previous regular-expression-based approaches. It is shown that SOL is capable of modeling very complicated functional transactions. Meanwhile, a translator is also developed to automatically convert a set of SOL-based transactions with the spec FSM into the corresponding functional coverage analyzer. The experimental results demonstrate that the proposed methodology can indeed improve the verification quality as well as increase the efficiency of regression verification. In a near future, we plan to develop a technique that can automatically and dynamically bias the pattern generator by instantly analyzing the functional coverage on-the-fly and then integrate this technique into our methodology.

References

[1] M. Keating and P. Bricaud, “Reuse Methodology Manual for System-On-A-Chip Designs, 3rd Edition,” *Kluwer Academic Publishers*, July 2002.

[2] J. Bergeron, “Writing Testbenches: Functional Verification of HDL Models, 2nd Edition,” *Kluwer Academic Publishers*, February 2003.

[3] D. Drako and P. Cohen, “HDL Verification Coverage,” *Integrated System Design Magazine*, pp. 46-52, June 1998.

[4] F. Fallah, S. Devadas, and K. Keutzer, “OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification,” *Proceedings of the Design Automation Conference*, pp. 152-157, June 1998.

[5] P. A. Thaker, V. D. Agrawal, and M. E. Zaghloul, “Validation Vector Grade (VVG): A New Coverage Metric for Validation and Test,” *Proceedings of the IEEE VLSI Test Symposium*, pp. 182-188, April 1998.

[6] B. Min and G. Choi , “ECC: Extended Condition Coverage for Design Verification Using Excitation and Observation,” *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, pp. 183-190, December 2001.

[7] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv, “User Defined Coverage - A Tool Supported Methodology for Design Verification,” *Proceedings of the Design Automation Conference*, pp. 158-163, June 1998.

[8] S. Asaf, E. Marcus, and A. Ziv, “Defining Coverage Views to Improve Functional Coverage Analysis,” *Proceedings of the Design Automation Conference*, pp. 41-44, June 2004.

[9] A. Ziv, “Cross-product Functional Coverage Measurement with Temporal Properties-based Assertions,” *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 834-839, March 2003.

[10] Y.-S. Kwon, Y.-I. Kim, and C.-M. Kyung, “Systematic Functional Coverage Metric Synthesis from Hierarchical Temporal Event Relation Graph,” *Proceedings of the Design Automation Conference*, pp. 45-48, June 2004.

[11] Y.-S. Kwon and C.-M. Kyung, “Functional Coverage Metric Generation from Temporal Event Relation Graph,” *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 670-671, February 2004.

[12] Y.-C. Yang, J.-D. Huang, C.-C. Yen, C.-H. Shih, and J.-Y. Jou, “Formal Compliance Verification of Interface Protocols,” *Proceedings of the IEEE International Symposium on VLSI Design, Automation, and Test*, pp. 12-15, April 2005.

[13] H.-M. Lin, C.-C. Yen, C.-H. Shih, and J.-Y. Jou, “On Compliance Test of On-Chip Bus for SOC,” *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 328-333, January 2004.

[14] K. Ara and K. Suzuki, “A Proposal for Transaction-Level Verification with Component Wrapper Language,” *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 82-87, March 2003.

[15] C. Browy, “Comparing TestWizard and Specman for Transaction-level Verification,” white paper, available at http://www.avery-design.com/twwp.html.

[16] H.-J. Schlebusch, G. Smith, D. Sciuto, D. Gajski, C. Mielenz, C. K. Lennard, F. Ghenassia, S. Swan, and J. Kunkel, “Transaction based design: Another Buzzword or the Solution to a Design Problem?,” *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 876-877, March 2003.

[17] Property Specification Language – Language Reference Manual, Ver. 1.1, http://www.eda.org/vfv/docs/PSL-v1.1.pdf.

[18] ARM Limited, *AMBA Specification (Rev 2.0)*, May 1999.