Efficient coverage analysis metric for HDL design validation

C.-N.Liu and J.-Y.Jou

Abstract: Simulation is still the primary approach for the functional verification of register-transfer level circuit descriptions written in hardware description language (HDL). The major problem of the simulation approach is to choose a good metric to gauge the quality of the test patterns. The finite state machine (FSM) coverage test can find most of the design errors in a FSM. However, it is impractical for large designs because of the state explosion problem. In the paper, a higher-level FSM model is proposed to replace the conventional FSM model in the coverage test. The state transition graph can be significantly reduced in the model so that the complexity of the test sets becomes acceptable, even for large designs. This higher-level FSM model, called the semantic finite state machine (SFSM) model, can be easily extracted from the original HDL code automatically with little computation overhead. The advantages of using this model instead of the conventional FSM model in HDL design validation are thoroughly discussed. The implementation results show that it is indeed a promising functional coverage metric.

1 Introduction

Due to the increasing complexity of modern circuit design, verification has become the major bottleneck of the entire design process [1]. Most verification efforts are often put on verifying the correctness of the initial register-transfer level (RTL) descriptions written in hardware description language (HDL). Manufacturing test is well-developed for testing real chips, however, the test is aimed at finding physical faults in the manufactured circuits and not suitable for verifying the functions of RTL designs written in HDL. Some formal verification techniques can verify the equivalence of a design across several different design levels [2], but they cannot verify the correctness of the initial HDL descriptions. Formal techniques for language containment and property checking [2] are making some progress on solving this verification problem. However, there is hardly any evidence that these techniques could offer comprehensive verification across a wide variety of designs. Thus, it appears that the simulation method will continue to play an important role in the verification process.

The major problem of the simulation approach is to choose a good metric to gauge the quality of the test patterns. One popular metric of verifying the design written in HDL is the code coverage metric used in software testing [3–5]. In most commercial tools, the HDL coverage metrics also include the software coverage metrics, for example, line coverage or path coverage. There are also a lot of other coverage metrics proposed for HDL, e.g. block coverage, event coverage, etc. [4–6]. Most of them are based on traversing the language structures completely. This can verify the correctness of each atomic action written in the HDL programs. However, besides the atomic actions, it is also important for the sequence of actions to be verified for performing the desired functions. It is difficult for these methods to find bugs related to the sequence of actions.

Although many different code coverage metrics have been proposed there is still not a single metric similar to the stuck-at fault model in the manufacturing test being popularly accepted as complete and reliable. Another approach is to verify the functionality by hardware concepts, such as the finite state machine (FSM) model. The FSM model is a well-studied model for describing a synchronous sequential machine. To verify the functionality of a FSM, a popular approach is to apply input patterns to traverse the whole state transition graph (STG) completely during the simulation process. It is often called the *FSM coverage* test [4, 5]. Because it traverses all the design space, it surely can find most of the design errors in a FSM. However, the sizes of the STGs for modern designs are often too large to be traversed completely. Using this approach to verify the behaviour of a FSM becomes impractical.

Since traversing the STG of a FSM exhaustively is considered as a confident method to verify the functionality of the FSM, this method can be practical if the STG can be reduced to a reasonable size. To reduce the state space, two abstraction techniques [7] are often used: controller extraction and many-to-one mapping. Controller extraction separates the datapath from the control circuits and verifies the remaining control parts. Because most design errors occur in the control logic [8], abstracting the numerous registers in the datapath can significantly reduce the tested state space without degrading the verification quality. Techniques have been proposed [8–11] for abstracting the datapath and extracting the controller in a design. All of them have good results on state space reduction such that the verification time for complex designs can be reduced.

IEE Proceedings online no. 20010203

DOI: 10.1049/ip-cdt:20010203

Paper first received 20th January and in revised form 12th December 2000

The authors are with the Department of Electronics Engineering, National Chiao Tung University, Taiwan, R.O.C.
E-mail: jimmy@EDA.ee.nctu.edu.tw, jyjou@bestmap.ee.nctu.edu.tw

Although the state space reduction is significant using the controller extraction technique, one may obtain further reduction by performing a many-to-one mapping on the resultant STGs. Many-to-one mapping means to map some states with similar properties in the original FSM into one state in the new FSM. In other words, it models FSMs at a higher level of abstraction. After this mapping, the sizes of the new FSMs are always smaller than or equal to the original ones. By carefully choosing the merged states, one can still have good test quality with fewer test cases. Although some techniques have been proposed to model FSMs at a much higher level of abstraction, such as the HLSM in [12] and the EFSM in [13], their mapping rules may not be suitable for verifying the functions of an HDL design. HLSM is proposed for performing job scheduling in the high-level synthesis. Therefore actions that are to be executed in the same functional unit are put into one state in the HLSM. However, if we traverse the HLSM model of an HDL design completely, errors between the actions in the same state may not be found because only one path is executed while passing this state. EFSM is proposed for verifying the functionality of an HDL design. It can cover every statement in an HDL design, which is the basic requirement in functional verification. However, its mapping rules are based on the state and input values. It is difficult to handle some FSMs such as the counters.

To meet the requirement of HDL functional verification, we propose another higher-level FSM model in this paper. This semantic finite state machine (SFSM) model can keep the FSM structure to verify the sequence of actions and use the semantic content of HDL descriptions to reduce the number of tested states dramatically. Therefore the benefits of both software coverage test (e.g. line coverage test) and hardware coverage test (e.g. STG coverage test) can be kept by using this model for verifying the functionality of an HDL design. In addition, because the mapping rules of this FSM model are based on the performed actions instead of the state and input values, one can have state reduction in more cases including the counters. By demonstrating with some examples, the advantages of using this model instead of the conventional FSM model in HDL functional verification can be clearly shown.

2 Semantic finite state machine model

Consider an 8-bit counter with synchronous load and reset functions. The behaviour of this counter is described in Verilog HDL [14] as shown in Fig. 1. Because the HDL can express the behaviours at higher level of abstraction, one can describe the counting behaviour by using a simple statement such as ***count = count + 1*** instead of using a case statement with 2^8 branches. Furthermore, because the HDL-based synthesisers have been in production use for many years, it is reasonable to assume that they are so robust that they hardly produce any 'buggy' circuits for datapath operators such as the plus operator. Therefore it is a better practice to simulate the descriptions of the same behaviour only once instead of verifying the same behaviour repeatedly in a large number of different states. This practice can greatly reduce the verification time without sacrificing any quality assurance.

```
module counter (clk, reset, load, in, count);
input        clk, reset, load;
input [7:0]  in;
output [7:0] count;
reg [7:0]    count;

always @(posedge clk) begin
  if (reset) count = 0;
  else if (load) count = in;
  else if (count = = 255) count = 0;
  else count = count + 1;
end
endmodule
```

Fig. 1 *Counter example written in Verilog HDL*

Our proposed *semantic finite state machine* (SFSM) model is developed based on the foregoing observation. We group explicit states with the same behaviour into one *semantic state*. The transitions between the explicit states in the same semantic state will be eliminated, and the transitions between different semantic states will be kept to make sure the FSM has the correct sequence of actions. Therefore the SFSM model is a reduced subset of the conventional FSM model. By using this modelling technique verification time can be dramatically reduced and the correctness of the sequence between actions can still be guaranteed. In the following Sections the SFSM model is clearly defined and explained.

2.1 SFSM model definition

Definition 1: A semantic finite state machine M is defined as the 7-tuple $\{S, V, I, O, U, A, T\}$, where

S is a set of semantic states
V is a set of state variables in the original FSM
I is a set of input symbols
O is a set of output symbols
U is a set of update functions for the state variables such that $U: S \times V \times I \rightarrow V$
A is a set of action functions for the outputs such that $A: S \times V \times I \rightarrow O$
T is a transition relation such that $T: S \times V \times I \rightarrow S$.

Definition 2: An update function is a block statement consisting of *assignments*, *logic* operations, and *arithmetic* operations only. The left-hand-side variable of this function is restricted to state variables. The right-hand-side variables of this function will be expanded recursively until they are constants, primary inputs, or state variables of the FSM.

Definition 3: An action function is defined similarly as the update function except the left-hand-side variable of this function is restricted to primary outputs only.

To explain the SFSM more clearly, we use a graph to represent it. The vertices of this graph correspond to the semantic states in the SFSM model. In each semantic state s, there is an update function $u_s \in U$ which updates the state variables of the original FSM. There is a directed arc t labelled as $f(v, i)/a_t$ from vertex p to vertex q in the graph if there is a transition of $T((p, v, i) \rightarrow q)$. The function $f(v, i)$ is called the *enabling condition* of this transition. The action function $a_t \in A$ updates the outputs of the original FSM while this transition occurs. We call this graph the *semantic state transition graph* (SSTG). The following Section gives an example of SSTG.

2.2 Example of SSTG

As an example we show in Fig. 2 the SSTG of the counter shown in Fig. 1. There are only three different behaviours in this counter. Therefore there are three semantic states in the SSTG. All the possible enable conditions are listed on the right of Fig. 2 with an unique label. The enabling condition of each transition is referenced with this label.

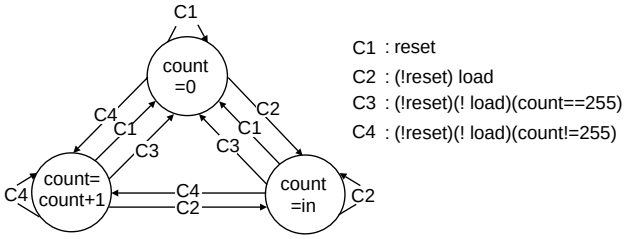

Fig. 2 *SSTG of counter example shown in Fig. 1*

Because there are no other outputs except the state variable ***count***, the action function of each transition is not shown in Fig. 2.

3 Automatic SSTG generation from HDL

The key idea of the SFSM model is obtained by the observation on HDL writing styles. Therefore we develop a simple algorithm to automatically extract the SSTG from the HDL. The extraction is done in five steps as follows:

3.1 Step 1: Generate the statement tree of the HDL FSM

The first step is to parse the HDL structure of the *synchronous section* of the FSM, which is the HDL code that describes the next-state and output equations of this FSM, to build the statement tree. The statement tree is clearly defined in definition 4. To explain the statement tree more clearly, we show a statement tree including the update functions in Fig. 3 for the FSM shown in Fig. 1.

Definition 4: A statement tree is a rooted, directed graph with vertex set N containing two types of vertices. A *nonterminal* vertex n has one or more children *child(n)* $\in N$. A mutually exclusive condition, which is composed of constants, the primary inputs, or the state variables of the FSM, will be attached on each edge from n to each child(n). A *terminal* vertex t, which represents the terminal block in the HDL code, has no children but a set of *update* functions U_t and *action* functions A_t.

While building the statement trees, we may find that some variables are not assigned in several program branches. During simulation, those variables will keep their original values in the unassigned branches, which acts like a self-assignment $x = x$. Therefore for the incompletely specified cases we automatically add self-assignments in the unassigned branches for those variables to represent the behaviours more clearly. In other words, each variable appearing in the statement tree will have its own update function or action function in the terminal vertices of this tree.

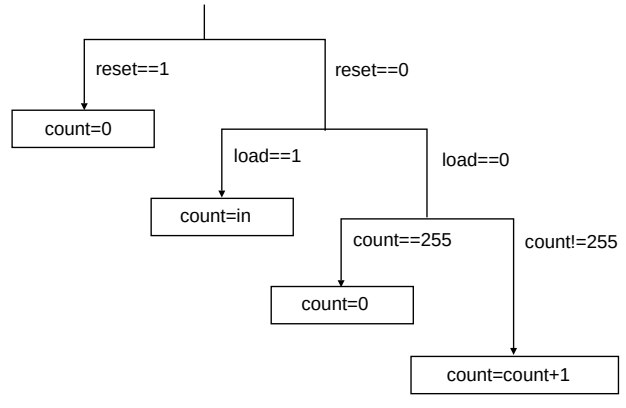

Fig. 3 *Statement tree for FSM shown in Fig. 1*

3.2 Step 2: Build the semantic states in the SFSM model

Definition 5: Two update functions or two action functions y, z are equivalent *iff* for each statement in y, there is a corresponding statement with the same left-hand-side variable existing in z and those two statements are identical.

While the statement tree of the HDL code is built the semantic states in SFSM model can be obtained directly from this tree. According to the definition of SFSM, we group the terminal nodes with the *equivalent update* functions into one semantic state. After checking all terminal nodes in the statement tree, all the semantic states can be built. The semantic state to which each terminal node belongs will be recorded in the data structure of the terminal node.

3.3 Step 3: Calculate the enabling condition of each transition

The product of the conditions on the path from the root to the terminal node is the *enabling condition* which enables the transition from some other semantic states to this target semantic state. In other words, each enabling condition represents a transition going to the particular semantic state corresponding to this terminal node. After traversing the tree structure of the statement tree, all possible enabling conditions can be calculated. These enabling conditions will be recorded in a hash table, *condition table*, with their associated terminal nodes.

3.4 Step 4: Build the transitions between semantic states in the SFSM model

Definition 4: An enabling condition e is contradictory to a semantic state s *iff* $\exists u \in u_s$ such that $e \cdot u \equiv \phi$.

For each semantic state, we can check each item in the condition table to find the enabling condition of each terminal node. Each terminal node will belong to a semantic state, thus we can find the target state of this transition through this terminal node. However, some transitions do not exist because of *condition contradiction*. For example, if the update functions of the current semantic state change the state variables to S0, and a transition going out from this state requires the state variables to be S1, this transition is absolutely impossible. After eliminating these contradictory transitions of each semantic state, a reachability analysis is performed on the SSTG to remove the states unreachable from the reset state. After all this, the final SSTG of this HDL FSM can be obtained.

4 Comparison with existing coverage metrics

4.1 Comparison with line coverage metric

Consider the simple FSM written in Verilog HDL shown in Fig. 4. It detects whether the *serial* input signal has a 101 sequence. If a 101 sequence is detected it sets the output *found* to 1. If the default action in line 18, which sets the output *found* to 0, is missing then this design will have faulty behaviour in some cases, explained as follows. First, the FSM is reset to its initial state S0. If we set the input *serial* to be {1, 1, 0, 1} for the next four consecutive clock cycles it will reach 100% line coverage for both faulty and fault-free designs. Applying these input patterns the output *found* has the same sequence, {0, 0, 0, 1} in both faulty and fault-free designs. That means this design error is not

```
1.  module fsm (found, serial, clk, reset);
2.  output        found;
3.  input         serial, clk, reset;
4.  reg           found;
5.  reg [1:0]     current_state, next_state;
6.  parameter [1:0]
7.     S0 = 0,
8.     S1 = 1,
9.     S2 = 2;
10.
11. always @( reset or serial or current_state) begin
12.    if (reset) begin
13.       next_state = S0;
14.       found = 0;
15.    end
16.    else begin
17.       next_state = current_state;
18.       found = 0;
19.       case (current_state)
20.          S0:begin
21.             if(serial = = 1) next_state = S2
22.          end
23.          S2:begin
24.             if (serial = = 0 ) next_state = S1;
25.             else next_state = S2;
26.          end
27.          S1:begin
28.             next_state = S0;
29.             if (serial = = 1) found = 1;
30.          end
31.       endcase
32.    end
33. end
34.
35. always @(posedge clk) current_state = next_state;
36. endmodule
```

Fig. 4 *HDL FSM written in Verilog HDL*

detected by these input patterns although the line coverage is 100%.

If we test this design based on the SSTG shown in Fig. 5, we have to set the input signals to be {R, 0, 1, 1, 0, 1, 1, 0, 0; 1, R, 1, 0, R} (R means *reset* = 1, 0 means *serial* = 0, 1 means *serial* = 1) for the next 14 clock cycles to traverse the whole SSTG from the initial state S0. According to these input patterns, the sequence of the output *found* is {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0} in the fault-free design and {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1} in the faulty design. Obviously we can find the undetected error in line coverage test according to our SFSM coverage test.

4.2 Comparison with FSM coverage metric

Consider the counter shown in Fig. 1. In the conventional STG, there are 256 states and 66 047 transitions in it. However, as shown in Fig. 2, there are only three states and 11 transitions in the SSTG. If we want to satisfy 100% FSM coverage in the conventional STG, it will take at least 66 047 clock cycles in simulation. In our SFSM model, only 13 clock cycles are needed to reach 100% SSTG coverage. Therefore the verification time can be dramatically reduced.

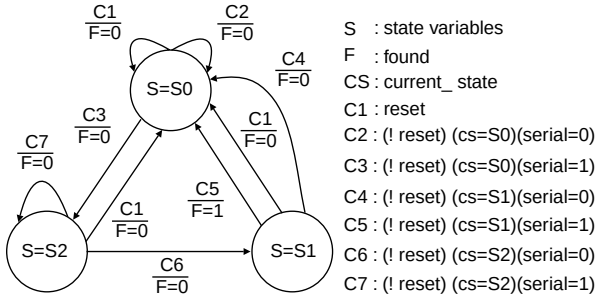

Fig. 5 *Final SSTG of HDL FSM shown in Fig. 4*

5 Implementation and experimental results

According to the steps described in Section 3, we have implemented the SSTG extraction tool ***SExt*** in C++ language. To conduct experiments we applied ***SExt*** to five FSM designs. The design statistics are given in Table 1. The column 'Lines' gives the number of lines of the original HDL code. Columns ***PIs*** and ***POs*** are the numbers of bits of primary inputs and primary outputs, respectively. Column ***SRegs*** gives the number of bits of all state registers.

The design ***bjk*** is the controlling FSM of a blackjack game machine [15]. According to the value of each card, it can decide to continue drawing a card, to hold cards, or go bust. The design ***count8*** is the 8-bit counter described in Section 2. The design ***count32*** is similar to ***count8***, but the number of bits is increased from 8 to 32. The design ***alarm*** [15] is a digital alarm clock operating on a 12-hour basis with separate AM/PM control. The values of the current time and the alarm time can be set from separate control pins. The design ***rankf*** is an 8×8 presorted rank filter. It keeps the last eight 8-bit data and puts them in a register array according to their ranks. We observe the corresponding data at the output by sending the desired rank to the input *Sel*.

In Table 2 we show the experimental results of applying ***SExt*** to those five designs, which are obtained on a 300 MHz UltraSparc II. The results give a good comparison between the sizes of the conventional STG and our SSTG. Even for the cases that the STGs are extremely large, such as the design count32 and rankf, the sizes of the SSTGs remain reasonable to be handled. Furthermore, according to the last column in Table 2, the computation time of SExt is quite small for all cases. It shows that the SSTG extraction process introduces almost no computation overhead.

Table 1: Design statistics

Design	Lines	PIs	POs	SRegs
bjk	163	9	8	4
count8	14	11	8	8
alarm	102	32	29	17
count32	14	35	32	32
rankf	553	13	8	88

Table 2: Experimental results of SSTG extraction

Design	STG states	STG edges	SSTG states	SSTG edges	CPU time, s
bjk	16	39	16	39	0.07
count8	256	66047	3	11	0.01
alarm	86400	$\approx 86400^2$	7	37	0.05
count32	2^{32}	$\approx 2^{64}$	3	11	0.01
rankf	$\approx 2^{88}$	$\approx 2^{176}$	65	4099	2.11

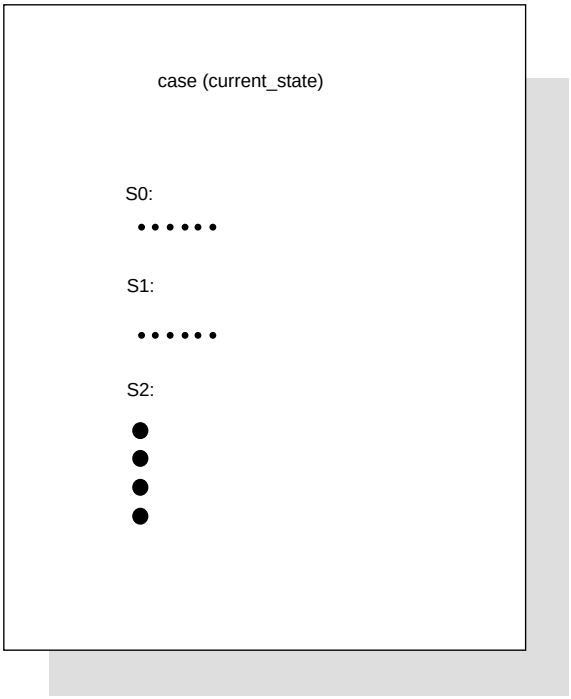

Fig. 6 *State-by-state writing style*

In Table 2 we find that the reduction on graph size is significant except the design ***bjk***. In this case, the HDL descriptions for the states in FSMs have been expanded as shown in Fig. 6. Because the main idea of the SFSM model is to keep the semantic states not be expanded into explicit states, not to merge several explicit states into one semantic state, the benefits of using the SFSM model in this case is thus not obvious. However, we also find that the sizes of the SFSMs are never larger than those of the conventional FSMs. In the worst case that designers describe every state in the FSMs by using state-by-state style, our SFSM model will have the same number of states and transitions as those in the conventional FSM model. The experimental results have shown that the sizes of our SFSM model are always less than or equal to the sizes of the conventional FSM model for those cases.

Another interesting observation can be obtained from the examples ***count8*** and ***count32***. When the number of bits of the counter increases, the number of states in the conventional STG grows exponentially, but the number of states in the SSTG remains the same. This is due to the fact that our SFSM model is built on a higher level of abstraction. The influences of bit numbers have been eliminated.

With the same designs we use another experiment to compare the line coverage metric against our SSTG coverage metric. The experimental results are shown in Table 3. For each design we generate test patterns in three different ways. The row 'directed for SFSM' gives the results with the carefully generated test patterns, which are generated such that the SFSM of each design is completely traversed. The design ***rankf*** has a large SSTG so that it requires a lot of patterns to traverse the SSTG exhaustively. Due to limited computation resource we stop the generation when the number of patterns exceeds 10 000. The row 'directed for LCM' gives the results with the carefully generated test patterns, which are generated such that the line coverage of each design achieves 100%. The row 'random' gives the results with random test patterns. For each random test set we continue generating patterns from a random seed until 100% line coverage is achieved or the number of patterns has exceeded 10 000. For each kind of test pattern the number of patterns generated is given in the row '#Pat' and the percentage of line coverage is given in the row 'LCM'. The rows 'SCM' and 'TCM' give the state coverage and transition coverage in the SFSM model, defined in definitions 7 and 8.

Table 3: Comparing coverage metrics

Statistics\Design		bjk	count8	alarm	count32	rankf
Directed	#Pat	111	13	54	13	10000
for	LCM	100%	100%	100%	100%	100%
SFSM	SCM	100%	100%	100%	100%	100%
	TCM	100%	100%	100%	100%	87%
Directed	#Pat	49	4	10	4	71
for	LCM	100%	100%	100%	100%	100%
LCM	SCM	100%	100%	100%	100%	71%
	TCM	67%	36%	27%	36%	1.7%
	#Pat	10000	2205	10000	10000	10000
Random	LCM	94%	100%	97%	75%	79%
	SCM	94%	100%	86%	100%	63%
	TCM	95%	91%	68%	82%	3.9%

Definition 7: State coverage metric (SCM) is defined as

$$SCM = \frac{number\ of\ semantic\ states\ visited}{total\ number\ of\ reachable\ semantic\ states}$$

Definition 8: Transition coverage metric (TCM) is defined as

$$TCM = \frac{number\ of\ transitions\ traversed\ in\ SSTG}{total\ number\ of\ reachable\ transitions\ in\ SSTG}$$

According to the results shown in Table 3, the line coverage always achieves 100% when the SCM and the TCM reach 100%. That means our SSTG test can completely cover the line coverage test, which is considered to be the basic requirement for HDL functional verification. In addition, from the data in the 'directed for LCM' row, we find that the TCM of each design is still low when line coverage achieves 100%. That means many transitions between actions are not traversed in the line coverage test. Some errors in the sequence between actions may still occur.

Furthermore, the numbers of patterns needed in the directed tests are always less than the numbers in the random tests. The coverage of random patterns is also lower than that of directed patterns. Therefore, in terms of achieving 100% coverage metric, we find the directed patterns to be superior to the random patterns for FSM designs. Even for the designs with large STGs such as ***count32*** and ***rankf***, we can still test them to achieve high coverage with a reasonable number of patterns.

6 Conclusions and future work

We have proposed a higher-level FSM model for HDL design validation. The STG can be significantly reduced in our model so that the FSM coverage test becomes acceptable even for large designs. The implementation results show that this higher-level FSM model, called semantic finite state machine (SFSM) model, can be efficiently extracted from the original HDL code automatically and is a promising functional coverage metric.

In the worst case that designers describe the FSMs in state-by-state style, the benefits of using current SFSM model may not be obvious because the main idea of the SFSM model is to keep the semantic states not expanded into explicit states, not to merge several explicit states into one semantic state. However, the state explosion problem should not occur for the man-made cases because it is very uncommon for designers to write an HDL program with state-by-state style for FSMs with hundreds of states. In the future, we will try to develop some merging operations to enhance the SFSM model such that we can handle the machine-generated HDL codes with state-by-state writing style. We may also try to build an automatic test bench generation system based on this coverage test. We believe that the functional verification of HDL descriptions can be greatly simplified with this automatic generator.

7 Acknowledgments

This work was supported in part by NOVAS Software Inc. and R.O.C. National Science Council under grant NSC89-2218-E-009-060. Their support is greatly appreciated.

8 References

1 EVANS, A., SIBURT, G., VRCKOVNIK, G., BROWN, T., DUFRESNE, M., HALL, G., HO, T., and LIU, Y.: 'Functional verification of large ASICs'. Presented at the 35th DAC, June 1998
2 VIS home page: http://www-cad.eecs.berkeley.edu/~vis
3 BEIZER, B.: 'Software testing techniques' (Van Nostrand Reinhold, New York, 1990)
4 DRAKO, D., and COHEN, P.: 'HDL verification coverage', *Integr. Syst. Design Mag.*, 1998 (http://www.isdmag.com/Editorial/1998/ CodeCoverage9806.html)
5 JOU, J.-Y., and LIU, C.-N.: 'Coverage analysis techniques for HDL design validation'. Presented at the 6th Asia Pacific conference on *Chip design languages* (APCHDL'99), October 1999
6 WANG, T.-H., and TAN, C.G.: 'Practical code coverage for verilog'. Proceedings of 1995 IEEE International Verilog HDL Conference. *IEEE Comput. Soc. Press*, Los Alamitos, CA, USA, pp. 99–104
7 GUPTA, A., MALIK, S., and ASHAR, P.: 'Toward formalizing a validation methodology using simulation coverage'. Presented at the 34th *DAC*, June 1997
8 MOUNDANOS, D., ABRAHAM, A., and HOSKOTE, V.: 'Abstraction techniques for validation coverage analysis and test generation', *IEEE Trans.*, 1998, **C-47**, (1), pp. 2–14
9 HO, C., and HOROWITZ, M.A.: 'Validation coverage analysis for complex digital designs'. Presented at the international conference on *Computer aided design*, November 1996
10 LIU, C.-N., and JOU, J.-Y.: 'A FSM extractor for HDL description at RTL level'. Presented at the 5th Asia-Pacific conference on *Hardware description languages*, July 1998, Seoul, Korea
11 LIU, C.-N., and JOU, J.-Y.: 'An Automatic Controller extractor for HDL descriptions at RTL', *IEEE Design Test Comput.*, 2000, **17**, (3), pp. 72–77
12 KUEHLMANN, A., and BERGAMASCHI, R.A.: 'High-level state machine specification and synthesis'. Presented at the international conference on *Computer design*, October 1992
13 CHENG, K.-T., and KRISHNAKUMAR, A.S.: 'Automatic functional test generation using the extend finite state machine model'. Proceedings of the 30th Design Automation Conference, 1993, Baltimore, MD, USA, pp. 86–91
14 THOMAS, D.E., and MOORBY, P.R.: 'The Verilog hardware description language' (Kluwer, Boston, MA, 1990)
15 SMITH, D.J.: 'HDL chip design' (Doone, Madison, AL, USA, June 1996)