UDC 681.32

YEV. YEF. SYREVITCH, A.L. KARASYOV, S.S. MEHANA

Kharkiv National University of Radioelectronics, Ukraine

FUNCTIONAL VERIFICATION QUALITY METRICS AT HDL-MODEL VERIFICATION

Principles of functional verification quality of digital devices models in hardware description languages are considered. The method of evaluation, which is based on quantity of checked functional modes, is offered.

verification, functional mode, hardware description languages

Introduction

During digital system design developers face the problem of verifying circuit description in HDL. Verification quality evaluation methods, which are accepted among software developers, are not suitable for projects in HDL. It is connected with HDL features, main of which are concurrency and signal presence in the description. The problem of verification quality estimation is as actual as test generation task during verification. Since verification using tests is spoken about, thus test quality is evaluated.

A set of different evaluation methods for test quality estimation exists [1 – 4]. For example, ad hoc metrics, program code coverage, state machine coverage, etc. The problem is that the majority of test quality metrics are mainly oriented onto controllability measure that is not difficult to calculate. But the fact of a code piece activation doesn't mean that it works correctly. The disadvantage of such metrics is that they don't give qualitative estimate of functional correctness.

User-defined functional coverage allows a developer, who has more detailed information about the project and implementation assumption, done at realization, to specify functional coverage for main project modes. An example, illustrating usage of functional coverage estimate during VHDL-model verification of the sectional microprocessor, is offered.

Analysis of test quality evaluation methods

Considered most spread methods of verification quality evaluation Special ad hoc metrics stand-alone. Ad hoc metrics include: error percentage, simulated sequence length after last found mistake, and general simulation cycle. Such metrics contain quantitative information, but produce very little qualitative information about verification or about percentage of project not being tested. Ad hoc metrics do not answer the question about verification successfulness, if the percent of errors aspire towards zero, but at that untested parts of the project exist.

The nest big subset of methods is program code coverage. They are:

1. *Line Coverage*. Measures, how many times the certain line in a code has worked (or has not worked) during simulation.
2. *Branches Coverage*. Measures, how many times the fragment of a code diverges into a unique flow.
3. *Path Coverage*. Measures, how many times the concrete path (switching operators and branching) is carried out during simulation.
4. *Expressions Coverage*. Measures controllability of a separate variable which influences calculation of target value of separately taken expression.

Ratings on a program code are based on a measure of controllability and allow revealing "holes" (that never worked at verification). Their lack is, that activation of the erroneous operator does not mean, that the mistake in a code will be shown on an observable point during simulation.

For overcoming the above-stated lack the metrics based on **a measure of observability** have been offered**.**

They show how well value on a variable is observable. The test giving 100% of line covering, achieves in average 77% at application of the metrics on observability.

One more disadvantage of ratings on a program code it that they do not give quality standard of testing for a functional correctness.

To overcome of the given lack it was offered to use the metrics based on **functional correctness** – a system of the supervising programs, allowing analyzing functional correctness and desirable events (for example, observable points) during simulation.

States and transitions coverage metrics in the graph is one more way of measurement of controllability. These metrics correspond to quantity of hits into this or that state or transition at performance of the test.

As well as program code coverage, states and coverage in a graph do not provide rating on observability and on a functional correctness.

Functionality coverage determined by a user allows the designer who possesses the greatest information on details of the low-level project and assumptions made at realization, to specify functional coverage for the important points of the project. Use of such metrics allows to draw a conclusion on how precisely the project corresponds to the specification.

Verification strategy

The strategy, used for verification of the given description, is based on a fact, that the designer for any input data can calculate reactions of the developed device, if data corresponds to modes (device states), described in the specification. Generalized steps of algorithm look as the following:

1. The verification engineer receives from the engineer - developer compiled hdl-code that describes a device or its logically (functionally) finished blocks.

2. Build tests on the received HDL-code by generation of sequences distinguishing a given operator and set control points.

3. Return received tests to the engineer - developer for calculation of corresponding target values and/or values in set control points.

4. Simulate generated tests on the HDL-code on order to get experimental values.

5. Compare the values received after simulating the hdl-code, and the etalon values received from the engineer - developer.

6. If they are not equal, a conclusion of mistakes, present in the design is made.

The algorithm of tests generation will consist in building of distinguishing test sequences for definition of the given functional element among the other of the given subset, driving tests into inputs of this element and transportation of results up to graph models external outputs or up to the nearest control points. Driving tests to a functional element and getting results is carried out on activated paths in the graph models. Path activization in the graph occurs similarly to path activization in D-algorithm for digital circuits, but language operators are used as primitives. Number of distinguishing test sequences equals to the number of operators in HDL-code.

In the listing 1 the fragment of VHDL description of the sectional multiprocessor KP1804BC1 is shown. This example was taken, because of its simplicity and typicalness.

Listing 1. Fragment of sectional microprocessor KP1804BC1 VHDL model

```
library IEEE; use IEEE.std_logic_1164.all; use IEEE.std_logic_signed.all;
ENTITY KP1804BC1 is port (DI: in std_logic_vector; AMC: in std_logic_VECTOR (0 to 1); ICPU: in std_logic_VECTOR (0 to 8); A,B: in INTEGER; Y : out std_logic_vector (0 to 3) ); END KP1804BC1;
architecture Data of KP1804BC1 is
signal Q : std_logic_VECTOR (0 to 3);
signal R,S,F : std_logic_VECTOR (0 to 3);
signal CI: std_logic;
type mem is array (0 to 6) of std_logic_VECTOR (0 to 3);
signal RAM:mem ;
BEGIN
process (A,B,Ci,ICPU,Di) begin
R_DECODER:   case ICPU(0 to 2) is
when "000"=>R <= RAM(A);
when "100"=>R<="0000";
when "111"=>R<=DI; when others=>
R<="XXXX"; end case;
S_DECODER:    case ICPU (0 to 2) is
```

```
when "000"=>S <=Q;
when "100"=>S<=RAM(A);
when"111"=>S<="0000";when others=> S<="XXXX";
end case;
ALU_OPERATION: case ICPU (3 to 5) is
when "000"  => F<= R+S;
when "100"  => F<=R and S ;
when "110"  => F<=R xnor S ; when others    =>
F<="XXXX"; end case;
OUT_FUNC: case ICPU (6 to 8) is
when "000"=> Y <=F;
when "010"=> Y<=RAM(A) ; when others    =>
Y<="XXXX"; end case;
Q_FUNC: if (ICPU (6 to 8)="000") then Q<=F;end if;
RAM_FUNC: if (ICPU (6 to 8)="010") then
RAM(B)<= F; end if; end process; END DATA;
```

The origin description is transformed into graph structure, which represents a composition of two graphs. First – information – describes dataflow and their conversion (similarly to an operational automaton in classical composite model with microprogram handle) without the registration of conditional branches. The second graph is developed as a network of conditions.

The example of graph model obtaining on a base of a code model is given below in the fig. 1.

There are two-place logic and arithmetic operations, and unary operations (as assignment, negation, and taking sign) in the informational graph.

Assignment operators (:=, <= accordingly) should also be included in the informational graph and tests should be building for them. But for simplification and volume decreasing these operations are not considered in the article.

Consider the algorithm of test generation for arithmetic operations distinguishing according to the principle “all from all” from the given subset. . Distinguishing sequences for synthesized arithmetic operators are submitted in fig. 2.

So that to distinguish operation "plus" from a subset {addition, subtraction, to increase, divide), it is necessary to submit a zero on one of inputs of the functional element, and on the other input – value, greater than one.

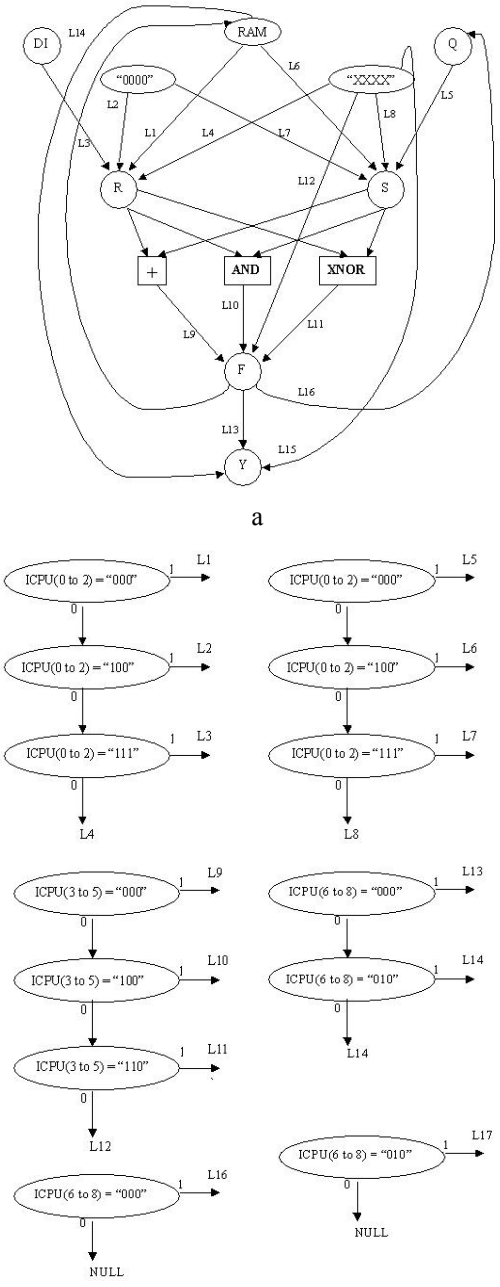


Fig. 1. Informational I-graph (a), control C-graph (b)

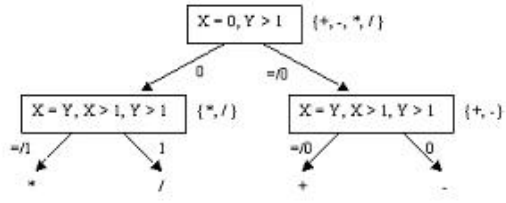


Fig. 2. Distinguishing sequences for arithmetic operators

Further we break up on subset (+, –) and (multiplication, division) by the following way: if the result of the previous step equals to ‘0’, then tested functional element is in the subset (multiplication, division); if not equal to ‘0’, then it is in the subset

(addition, subtraction). Then drive equal values greater than ‘1’ onto both inputs. If the result is not ‘0’, the functional element is ADDITION.

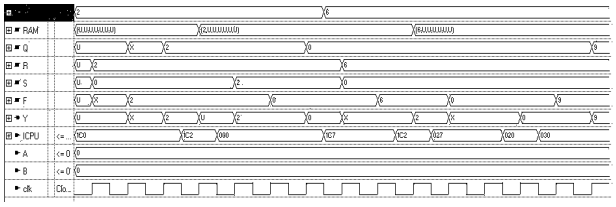
In the fig. 3 a table of test vectors generation and modes correspondence is given.

DI	RAM	Q	R	S	F	Y	ICPU <0-8>	R/S DECODER	ALU OPERATION	OUT FUNCTION	OPERATION UNDER TEST	ETALON
2	X	X	2	0	Y	X	111_000_000	R<= DI S<= 0000	F<= R+S	Y<= F	+	Y<= 2
2	X	X	2	0	2	X	111_000_000			Y<=RAM(A)		
2	X	2	2	0	2	2	111_000_000					
2	2	2	2	0	2	X	111_000_010	R<=RAM(A) S<=Q				
2	2	2	2	2	2	2	000_000_000			Y<= F		Y<= 4
2	2	4	2	2	4	4	000_000_000					
6	2	4	6	0	4	X	111_000_111	R<= DI S<= 0000	F<= R+S	Y<="XXXX" Y<=RAM(A)	AND	Y<= 4
6	2	4	6	0	6	X	111_000_111					
6	6	4	6	0	6	2	111_000_010					
6	6	4	6	4	0	X	000_100_111	R<=RAM(A) S <= Q	F<= RandS	Y<="XXXX" Y<= F		
6	6	4	6	4	4	X	000_100_111					
6	6	4	6	4	4	4	000_100_000					
6	6	4	6	4	13	4	000_110_000	-//-//-//-	F<=RxnorS	-//-//-//-	XNOR	Y<=13
6	6	13	6	4	13	13	000_110_000	-//-//-//-		-//-//-//-		

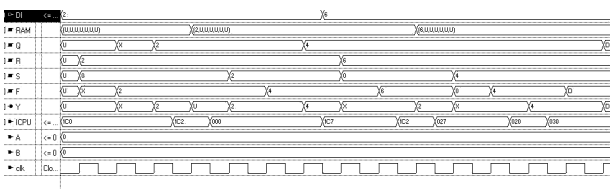
Fig. 3. Test obtaining

In the fig. 3 steps of test generation are described for verification of three modes: addition, logic AND, and XNOR. Each mode corresponds to a certain set of input restriction vectors, called restriction vectors, obtained during test generation. Number of vectors depends on number of control points and number of vectors, necessary for providing

Functional mode is a function, which is executed by a device on input data transformation Modes of moving and reading/writing in the given example are not functional modes, but are providing for them.



a



b

Fig. 4. Simulation of the code , which contains (a) and doesn’t contain design error (b)

A diagnostic experiment stays in the following: a design error of SUBSTITUTION type is inserted into the code.

Simulation is carried out using input patterns, generated on the graph structure. Further two figures are represented: the first (fig. 4, a) contains error-free code simulation results (modes in the specification and in the code coincide);the second – result of code simulation with inserted error (fig. 4, b). It is obvious, that results at Y-output, which is a control point, differ from each other.

Test quality metrics

On a base of functional modes mode graph is build, bypassing of which gives the ides of verification quality. Specification for a microprocessor is set in informal, language form. For example, it looks like the following:

The field of the microcommand contains 9 bats. Three least significant bits determine operands, three middle bits - an operation, three most significant - a code of a receiver and RAM operation. If the code of operation is "000", then **F=R+S** is carried out ;if "100" **F=RandS** is carried out; "110" corresponds to **F=RxnorS**. The code of operands "000" sets R=RAM (A) and S=Q; the code "100" corresponds to R = "0000" and S=RAM (A); and "111" sets R=DI and S = "0000"; the code of the receiver describes the following: if it is equal "000", then Y=F and Q = F; if "010", Y=RAM (A) and RAM (B) =F is carried out.

A microcommand from the specification corresponds to some functional mode.

As it has been said above, sets of vectors of restrictions also correspond to some operating mode of DD. Thus a sequence of restriction vectors allows setting a way of graph bypassing.

For the given example of the sectional microprocessor a way of bypassing is the following: ‘+’ –‘ AND’ –‘ XNOR’.

Consider the problem of obtaining at least one decision at primary inputs. Assume, that all primitives are functional. Besides, functional primitives (built-in operators) allow obtaining reactions for all possible values of operands ranges of definition. At that during propagation through a functional element there is at least one pair of input values, providing the given output (for operations «+» and «–» it is obvious; for «*» and «/» for first input – ‘1’, for second – output value. It is fair for inputs, which do not have predefined values. At absence of restrictions on lines up to considered functional element there is always a

decision – values on graph inputs, which allow to propagate necessary distinguishing sequence.

Consider the situation, when a restriction vector, obtained from the verification engineer, does not correspond to any functional mode. If it is impossible to calculate etalon reactions for the vector, obtained on external inputs, then it is necessary to make up a decision about either mistaken code, or incorrect – written test. Test is generated correctly, if during justification of distinguishing sequences for the given functional element at least one decision exists on external inputs and this is fulfilled, as considered earlier. Then a conclusion is done about mistaken code. The admitted place of mistake is control constructions.

It is possible to come out with the situation, when some restriction vector, not being belonged to the specification, nevertheless (according to the designer's words) corresponds to some mode. If the designer (or any ideal external model) can provide etalon reactions for this mode, then the specification dynamically is extended and expanded.

The diagram of code, specification, and test correspondence is given below in the fig. 5.

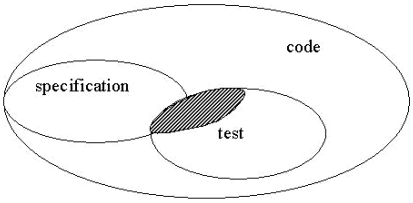


Fig. 5. Test, specification and code-description correspondence

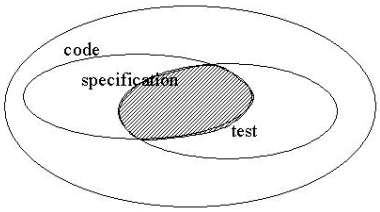


Fig. 6. Test, specification and code-description correspondence after extra modes addition

After inclusion additional functional modes into the specification the diagram looks as follows.

The test covers now more of the specification, than in the first figure.

Proceeding from all aforesaid, we shall define a quantitative measure of test quality.

$$Q = \frac{f_{code}^{ver}}{f_{spec} + f_{extra}} * 100\% , \qquad (1)$$

where f_{spec} – modes from the specification; f_{extra} – additional modes inserted into the specification; f_{code}^{ver} – modes, verified in the code. Using this evaluation allows giving quantitative equivalent of qualitative measure.

Conclusion

Scientific value and novelty of the offered methodology is in using functional coverage metrics for functional verification quality evaluation. The example was suggested that illustrates usage of functional coverage metrics during functional verification of VHDL description of sectional microprocessor KP1804BC1.

Literature

1. Tasiran S., Keutzer K. Coverage Metrics For Functional Validation Of Hardware Designs // IEEE Design & Test Of Computers, July-August 2001. – P. 36-45.

2. Kryvulya Gennadiy, Syrevitch Yevgeniya, Karasyov, Andrey Chegikov Denis. Test Generation for VHDL Descriptions Verification //Proceedings of IEEE East – West Design & Test Workshop. – Odessa, Ukraine, September 15-19. – 2005. – P. 191-195.

3. Рустинов В.А., Сыревич Е.Е., Сыревич А.В., Чегликов Д.И. Процедуры импликации на арифметической операциях при синтезе тестов верификации // АСУ и приборы автоматики. Всеукраинский межведомственный н.-т. сборник. – Х., 2005. – Вып. 130. – С. 4-13.

4. Foster Harry D., Krolnik Adam C., Lacey David J. Assertion-Based Design Kluwer. – Academic Publishers, USA. – 363 p.

Поступила в редакцию 14.03.2006

Рецензент: канд. техн. наук, проф. Н.Я. Какурин, Харьковский национальный университет радиоэлектроники.