Simulation-based validation of VHDL descriptions using constraints logic programming

Christophe Paoli, Marie Laure Nivet, Fabrice Bernardi, Laurent Capocchi

▶ To cite this version:

Christophe Paoli, Marie Laure Nivet, Fabrice Bernardi, Laurent Capocchi. Simulation-based validation of VHDL descriptions using constraints logic programming. IEEE Workshop on RTL and High Level Testing, Nov 2004, Osaka, Japan. pp.S2.3. hal-00440834

HAL Id: hal-00440834
https://hal.science/hal-00440834

Submitted on 11 Dec 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Simulation-based validation of VHDL descriptions using constraints logic programming*

Christophe Paoli Marie-Laure Nivet Fabrice Bernardi Laurent Capocchi

UMR CNRS 6134, University of Corsica,
SPE – Systèmes Physiques pour l'Environnement,
Campus Grossetti, Bat 018, BP 52, 20250 Corté, France.
[cpaoli|nivet|bernardi|capocchi]@univ-corse.fr

Abstract

This paper presents a simulation based validation approach for test vectors generation. We suggest to borrow techniques used successfully in the software testing and constraints logic programming areas. Our methodology is based on the three following steps: VHDL code modeling and analysis, constraints-based stimuli generation and test sequences generation.

1. Introduction

The verification that no error was introduced between the algorithmic level and the RTL level remains an important issue which requires an automation. We know this field of study under the name of functional validation or functional verification. IEEE [1][2] defines verification as the means of establishing the correspondence between a product and its specification, and validation as the means of insuring that a product achieves the function for which it was conceived. In the electronic designer and tester community, the term of verification, that is formal verification, refers to the methods consisting in proving that a circuit will behave as it is supposed to, whereas the term of validation, so called simulation based validation, refers to the methods consisting in exciting a description of circuit by a series of stimuli: the set of stimuli and the observed values is called test vectors.

In spite of recent progress, the formal verification which proposes to prove mathematically the correctness of a description is only realizable for small descriptions.

Indeed the automation of the method implies the exhaustive analysis of a very large space of inputs and is thus restricted to reduced parts of a description. So the simulation based validation is still the best method for design verification. The reader can refer to the various articles published in [3] which present the last projections in formal verification.

Our interest here is in the approaches of the simulation based validation. The production of the test vectors is one of the principal problems of the simulation based validation. The disadvantages of this approach are that an exhaustive simulation is necessary to guarantee the exactitude of a description, and that the complexity of current descriptions makes this task unfeasible. The random generation of test vectors is relatively easy but does not guarantee the verification of all the functionalities of the description of the circuit. In general, the designers of circuits generate manually the test vectors according to their knowledge of circuit functionalities. The problem of the automatic generation of test vectors thus remains a subject largely studied by the scientific community.

In this paper, we suggest an approach of test vectors generation which borrows techniques used successfully in the software testing area and which uses a simulator developed within our laboratory. We present in the following section a state of art - which is not meant to be exhaustive - of the automatic test vectors generation. In the third section, we present our methodology to generate the test vectors. Results are presented in section 4 and the prospects are given in section 5.

2. State of art

Our intention is not to describe in details all the existing methods derived from the software testing area, but simply to present the most representative, to show their limits, and give the reasons which led us to develop our methodology.

Software testing is usually done at several levels. They are commonly referred to as unit testing, integration

* A part of this research is founded by CTC (Collectivité Territoriale de Corse) under the contract number A-03-0842; Project 15-3740 "Développement d'un outil de test de circuit et amélioration de la testabilité des circuits" (2003-2004).

testing, and system testing. Theirs objectives are respectively:

- to verify that individual units (the smallest compilable components) function correctly
- to test the integration of components and the communication between them
- to find defects that are attributable to the behavior of the system as a whole, rather than the behavior of individual components, and to test the software functions as a complete system.

Whatever the level of development, the activity of test generally proceeds in two times:

- selection of a set of input data (often called test set or test data)
- execution of the software with these input data and observation of its behavior. This observation is often carried out by a human being which must pronounce an "oracle" (i.e. it must decide if the results of the execution are correct).

Our study is at the unit level because we consider that a behavioral VHDL program is a module which can form part of a larger program. Concerning oracle, VHDL language [4] envisages the use of files of simulation, called test-bench, which make it possible to check and announce any error met during simulation.

The quality of the test data can be evaluated using metrics. The software metrics describe the properties of the code of the software. Their objective is to evaluate if the requirements for quality are satisfied during all the phases of the software life cycle. The metrics are measurements which make it possible to evaluate the complexity of a software and thus offering a help for the test.

The use of coverage metric [5] constitutes an approach to test vectors generation. In [6][7], the test vectors generation is based on observability and instructions coverage metric[8]. Tags are associated to each variable assignment in the RTL level Verilog description and must be propagated, as for the behavioral faults, towards an exit during simulation. The vectors are generated thanks to a hybrid algorithm SAT (problem of satisfaction of Boolean expression).

The method developed by Kapoor and Armstrong [9] takes into account VHDL descriptions of RTL level and aims at the detection of design errors. It uses the CDFG (Control and Data Flow Graph) defined in [10] and generates test vectors starting from the instructions coverage criterion. Symbolic values are used to represent the signal values and their transitions. These values are affected by the user to justify the paths through the CDFG which make it possible to respect the instructions coverage criterion. A test-bench is finally generated in order to simulate description.

In [11] and [12] execution paths for which one wishes to generate test vectors are specified through certain annotations, managed by the user, in VHDL behavioral description. Each annotated path is translated into constraints, a constraints solver is then used to produce the test vectors which allow their execution. The test vectors are finally converted into format WAVES [13] to facilitate the automatic generation of test-bench.

The methods of [14] and [15] use concepts borrowed to the formal verification to extract from a Verilog description, a Finite State Machine (FSM), from which the test vectors are generated. Whereas the method of [14] is based on the path coverage, the one developed in [15] is based on the transitions (change of state) coverage.

The approaches of [16] and [17] take into account VHDL descriptions of RTL level. The first approach combines techniques used in the software testing area: instructions coverage criterion starting from a STG (State Transition Graph); and techniques used at the gate level (RTL synthesis tool). The generated test vectors make it possible to detect faults of low level, whereas the approach developed in [17] aims at detecting design errors. This approach uses a genetic algorithm, which interacts with a VHDL simulator, and automatically generates test vectors respecting the branch coverage criterion.

The interested reader can refer to the various articles published in [18] which present the last projections as regards functional verification and generation of test-bench. The methods previously described can be the subject of criticisms and the following comments. In [11] and [12] the designer must annotate the number of times that an instruction of control must be carried out. This implies a knowledge on the functionality of description and can give paths whose execution is impossible. However, an effective model of constraints was defined which makes it possible to model VHDL instructions crossed by a path. We used this model in our approach. The methods of [17][6][7][16], take into account HDL descriptions of RTL level which is lower than the algorithmic level. Moreover the approach of [16] is dependent of a synthesis tool. In [15] and [14], the developed methods are applicable only for relatively small designs because of the state space size of the FSM, in spite of the improvements made in [14].

To conclude this section, we can say that the static analysis of the code is not sufficient. As a matter of fact, in the presence of loops, the number of execution paths are potentially infinite. Furthermore some of them can be unfeasible. We think that a solution is to connect a simulator to the test vectors generator and thus evaluate the quality of the generated test vectors with pertinent metrics.

3. Our approach

In this section, we present our methodology to generate the test vectors. Our approach is limited to a subset of VHDL defined in [19].

3.1. Overview

As the various methods of software testing, the test of white box type seems most suitable for behavioral VHDL descriptions. This testing technique uses the structure of control of the program, i.e. the code, as a basis to develop or evaluate tests. When this strategy is used the program is seen as a white box in opposition to the test of the black box type which does not use any information on the structure of the coded program. By using this strategy, the tester selects the test data by the examination of the logic of the program. This approach is based on a graph of the program called control flow graph. A multitude of criteria of selection defined on this graph were suggested. The principal methods are the coverage of instructions, the coverage of branches and the coverage of paths. The coverage of instructions requires the execution of each instruction in the program at least once during the test. The coverage of branches requires that each branch of the program will be crossed at least once. A branch is a point in the program from which one or more sets of alternatives of program instructions are selected. The coverage of paths is the most rigorous method and most effective and consists in crossing all the paths which respects a certain criterion. These methods can be classified by means of the relation of inclusion: a criterion X is included in the criterion Y if any test set satisfying Y also satisfies X. One obtains that the coverage of instructions is included in the coverage of branches, it even included in the coverage of paths. However, the coverage of 100% of the paths is theoretically impossible to reach. As we said earlier, in the presence of loops, the number of paths of a control flow graph is potentially infinite. For this reason, several other criteria were defined with the aim of filling the gap between the branches coverage and the paths coverage. In this way we suggest to use a powerful test criterion called the structured testing criterion defined by Watson and McCabe [20]. In order to find data to apply to input port (stimuli) of the VHDL description we chose to represent paths, i.e. the crossed VHDL instructions, by systems of constraints.

The basic operations (cf. Figure 1) of our test vector generator approach consist in the three following steps: (i) VHDL code analysis, (ii) constraints-based stimuli generation, (iii) test sequences generation.

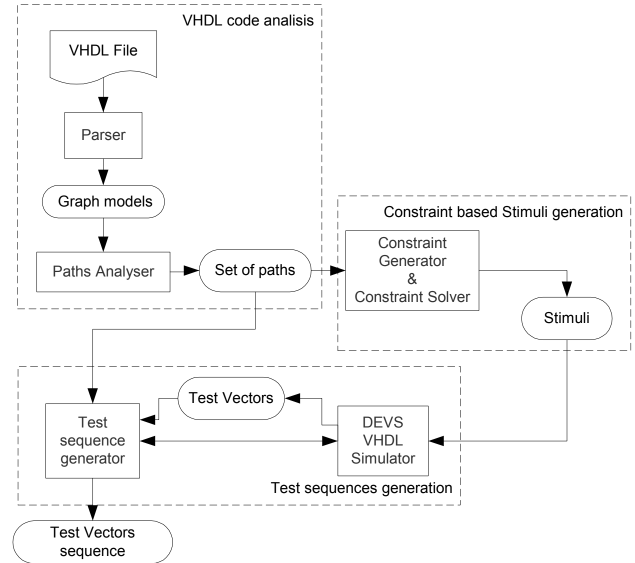

Figure 1. Global view of our approach

These steps are detailed in the three following sub-sections.

3.2. VHDL code analysis

The first part of our approach concerns VHDL code analysis. The goal of this step is to

- generate the graphs needed by the structured testing criterion
- generate and analyze the produced paths

The structured testing criterion uses the cyclomatic complexity [21] as an index of the number of paths to be tested in a software module. This criterion is stronger than other common coverage criteria like statement coverage or branch coverage. The cyclomatic complexity, also called v(G), is a software metric based on the control flow graph (CFG). This complexity represents the number of linearly independent paths (called a “basis set”) of the CFG. It is defined for each module by e-n+2, where e and n are respectively the number of edges and nodes in the CFG. These paths can be used to construct any other existing paths. The problem of finding an appropriate set of v(G) paths is solved using a powerful algorithm: the Poole’s algorithm [22]. The basis generated with such an algorithm will be referred to as *primary basis*.

Behavioral VHDL programs contain multiple communicating concurrent processes, delta delay mechanism for signal assignments and the concept of time which are not found in traditional software programming languages. Variables and signals keep their value through time and are not directly controllable as input ports. These VHDL features imply that some paths of the *primary*

basis can not be exercised directly by their corresponding test vectors.

We have classified the paths of the *primary basis* in three sub-sets:

- The paths that need an event on an internal signal (any change in the current value of a signal). These paths of the basis do not start at the entry of the program. They must be combined with a path of the primary basis which starts at the entry of the program. We define this type of path as *path to be modified*.
- The paths that traverse a decision node that implies a variable or an internal signal assigned in an other path. A decision node models any VHDL control instruction (e.g. an if statement). We define this type of path as *path to be scheduled*. We have to construct a sequence of test vectors. This sequence must be executed before the test vector that exercises the *path to be scheduled*.
- The path that can be exercises directly, because they do not contain references to internal signal or variable. They can be considered as the entry of the VHDL program. All sequences have to begin with one of these paths, we called *primary paths*.

In order to analyze the *primary basis* and to determine these three sub-sets, we use the Dependence Flow Graph (DFG). This graph is defined on the same nodes as the CFG. The edges of the DFG represent the dependencies between the program's statements. We have developed in [19] algorithms which generate the list of solution paths that allows to change the *paths to be modified* and the list of solution paths that allows to construct the sequences for the *paths to be scheduled*. All of them must begin with a *primary path*. We choose to keep all the solutions in a single list of solution paths. For each path many stimuli are possible and the choice between them is discussed in the section 3.4.

3.3 Constraints-based stimuli generation

The second part of our approach concerns the test data generation. We have to find data to apply to input port (stimuli) of the VHDL description. We chose to represent paths as a constraints system.

Vemuri and Kalynaraman [12] have defined a method of test vectors generation for VHDL programs using a constraints model which have inspired our work. Their constraints model allows to take into account the VHDL delta delay mechanism for signal assignments and the concept of time which are not found in traditional software programming languages.

More recently, [23][24] have also suggested to solve the problem of the test generation using constraint programming. Let's start with some definitions.

A Constraint Satisfaction Problem (CSP) consists in:

- a set of variables $X=\{ x_1, \dots , x_n \}$
- for each variable x_i, a finite set D_i of possible values (its domain)
- and a set of constraints $C=\{ C_1, C_2, \dots , C_m \}$ restricting the values that the variables can simultaneously take.

A solution to a CSP is an assignment of a value from its domain to every variable, in such a way that every constraint is satisfied. We may want to find:

- just one solution, with no preference as to which one
- all solutions
- an optimal, or at least a good solution, with a given objective (e.g., cost) to be maximized or minimized

The constraints can be solved by using a Constraints Logic Programming (CLP) language including a solver of constraints. If all the domains of the variables are finite and can be enumerated, then the constraints solver will find a solution if it exists. In fact, if the solver of constraint uses the concept of backtracking, so all the solutions can be obtained. Let us note that the force of a environment containing constraints is that the problem can be partially specified, leaving the environment computing the value of the variables.

In the context of test vector generation, i.e. the generation of data input from paths, we have two elements to model:

- the translation of VHDL objects (variable and signals) into constraint variables, and the definition of their domain
- the translation of VHDL instructions into constraints

We have developed algorithms in [19] which translate paths into a system of constraints composed by:

- domain constraints which define the domain of the variables and signals,
- relational constraints due to the statements crossed by a given path.

After the translation into a constraint programming language [25], we used the OPL studio solver to obtain all the solutions of the constraint system. We thus obtained a set of stimuli for each v(G) paths.

3.4. Test sequences generation

The third part of our approach concerns the test sequences generation. As explained in section 3.2 some paths, the *paths to be scheduled*, do not only need a stimuli but a sequence of stimuli. They can not be directly exiting, but are dependant of the execution of an other path. In order to find the valid sequence, we need to build a schedule graph. This graph represents the dependencies between the paths including notion of priority in execution.

As a result of the schedule graph analysis, we select a list of linked paths and for each path a specific stimuli. In order to validate the produced sequences, we connect a VHDL simulator developed in our laboratory: the BFS-DEVS behavioral fault simulator [26]. This tool allows to simulate faulty definition but can also be used as a classic VHDL simulator. We use it in order to compare values of non controllable VHDL objects (internal signal and variable): the value obtained from the simulator and the one obtained from the solver. If these values are not equal, we compose another sequence according to the schedule graph until we find a solution. If there is no valid solution in the stimuli set, the path can not be excited.

4. Results

We have implemented the two first steps of the test vectors generator in Prolog and LISP languages [19][27][28]. The Table 1 shows the first results we obtained on descriptions of ITC benchmarks [29].

Table 1. Results on ITC benchmarks.

	#l	#p	#I	#o	#v(G)	#pp	#ps	#pm
B01	111	1	4	2	19	3	16	0
B02	71	1	2	1	13	3	10	0
B03	142	1	6	1	19	3	16	0
B04	103	1	6	1	12	3	9	0
B05	333	3	3	6	52	3	12	37
B06	129	1	4	4	17	3	14	0
B07	93	1	3	1	15	3	12	0
B08	90	1	4	1	11	3	8	0
B09	104	1	3	1	11	3	8	0
B11	119	1	4	1	23	3	20	0
B13	297	5	5	7	56	14	42	0
B14	518	1	34	54	164	3	161	0
B20	1040	3	34	22	329	5	324	0
B22	1547	4	34	22	492	7	485	0

The first column corresponds to the names of the VHDL descriptions. We did not take into account from B16 to B19 and B21 descriptions cause they are structural types. The following four columns are related to programs features: number of lines (#l), number of processes (#p), number of input ports (#i) of output ports (#o). The other columns present the results obtained with our software after the first step that is code analysis: the cyclomatic complexity number (#v(G)) which represents the total number of paths to be exercised respecting the structured testing criterion, the number of primary paths (#pp), the number of paths to be scheduled (#ps) and the number of paths to be modified (#pm).

A quick analysis of these figures gives knowledge about the structure of the VHDL code and the way the processes are connected. For example in the case of the b22, the number #pm equal to zero shows that the processes are no connected as opposite to the b05 description.

In order to illustrate the second and the third step of our approach, we choose to focus on the b02 VHDL description. The results obtained after constraints generation and schedule graph analysis are shown in Table 2.

Table 2. Focus on b02 VHDL description

Path name	# stimuli couples	solution paths
1	16	-
2	16	-
3	4	5-8
4	4	5-8
5	8	10
6	8	7-4
7	8	9
8	4	11
9	4	11
10	4	6-12
11	4	6-12
12	8	13-3
13	24	-

The first column corresponds to the v(G) paths needed to test the b02 description: path 1 to path 13. The second column gives for each path the number of the stimuli couples obtained after the resolution of the constraint system. The last column presents the list of the solution paths that allows to construct the sequences for the *paths to be scheduled.* For example in the case of the path 12: eight couples of stimuli are possible. But a sequence has to be exercised before. This sequence has to be composed with stimuli couples of the path 13 or with stimuli couples of the path 3. If the primary path 13 is chosen, the number of stimuli couples that are possible is equal to 24.

The implementation of the third step, i.e. test sequences generation, is not finished. We have not made for the moment the link between the stimuli test generator and the VHDL simulator, but the results obtained seem already effective.

5. Conclusion and futures works

We have presented in this paper the test vectors generation problem in terms of executing a particular set of paths. We have used software testing techniques and constraint logic programming.

We have developed a software prototype we called GENESI for GENErator of StimulI. We must finish its implementation, particularly the test sequences generation step. But it already permit to help designers and testers to find valid test sequences.

In order to validate our approach, we feel the need for benchmarks of algorithmic level VHDL descriptions and the definition of a standard for a subset of VHDL for behavioral specification.

Our future investigations will concern:

- the study of different metrics for path or stimuli selection and measure the quality of the test sequences
- the design improvement of high level VHDL descriptions using graphical HDL tools, e.g. visualizing paths coverage on the graph
- the consequences of the writing quality of VHDL code on synthesized electronic circuit.

References

[1] IEEE Press. IEEE Standard Glossary of Software Engineering Terminology; ANSI/IEEE Standard 729-1983; 1983.

[2] IEEE Press. IEEE Standards Collection, Software Engineering, 1994.

[3] IEEE Press. Formal Verification of Commercials ICs. IEEE Design & Test of Computers, v. 18, no. 4, New York, July-August 2001.

[4] IEEE Press. VHDL, Language Reference Manual, IEEE Standard 1076, 1987.

[5] B. Beizer, Software Testing Techniques, Van Nostrand Rheinhold, New York, second edition, 1990.

[6] F. Fallah, S. Devadas, and K. Keutzer, Functional vector generation for HDL models using linear programming and 3-satisfaiability, Proc. ACM/IEEE DAC, pp. 528–533, 1998.

[7] F. Fallah, P. Ashar and S. Devadas, Simulation Vector Generation from HDL Descriptions for Observability Enhanced Statement Coverage, Proc. Of 36th ACM/IEEE conference on Design automation conference, New Orleans, LA USA, June 21 – 25, 1999.

[8] F. Fallah, S. Devadas, and K. Keutzer, OCCOM: Efficient computation of observability-based code coverage metrics for functional verification, Proc. ACM/IEEE DAC, pp. 152–157, 1998.

[9] S. Kapoor, J. R. Armstrong, S. R. Rao, An Automatic Test Bench Generation System, Proc. VHDL International Users Forum, pp. 8-17, 1994.

[10] C. H. Cho and J. R. Armstrong, B-algorithm: A Behavioral Test Generation Algorithm, Proc. ITC, pp. 968-979, October 1994.

[11] R. Kalyanaraman, Behavioral Test Vector Generation in VHDL/WAVES Environment, M.S. Thesis, 1993.

[12] R. Vemuri, R. Kalyanaraman, Design Verification tests from Behavioral VHDL Programs, IEEE Trans. on VLSI, Vol. 3, N° 2, pp. 201-214, June 1995.

[13] IEEE Press. Standard 1029.1-1991, IEEE Standard for waveform and vector exchange (WAVES) VHDL, September 1992.

[14] J. Shen and J. A. Abraham, Verification of Processor Microarchitectures, Proc. IEEE VLSI Test Symposium, pp. 189-194, 1999.

[15] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architecture validation for processors, Proc. Int. Symp. Computer Architecture, pp. 404-413, 1995.

[16] E.M. Rudnick, R. Vietti, A. Ellis, F. Corno, P. Prinetto, and M. Sonza Reorda, Fast Sequential Circuit Test Generation Using High-Level and Gate-Level Techniques, Proc. DATE, pp. 570-576, 1998.

[17] F. Corno, M. Sonza Reorda, G. Squillero, A. Manzone and A. Pincetti, Automatic Test Bench Generation for Validation of RT-Level Descriptions: An Industrial Experience, Proc. IEEE DATE, March 2000.

[18] IEEE Press. Functionnal Verification and Testbench Generation. IEEE Design & Test of Computers, v. 21, no. 2, New York, March-April 2001.

[19] C. Paoli, PhD in Computer Science, Thesis title : « Validation de descriptions VHDL fondée sur des techniques issues du domaine du test de logiciels » (Validation of VHDL descriptions based on software testing techniques), defended at the University of Corsica the december 20th 2001.

[20] A.H. Watson, T.J. McCabe, "Structured Testing: A testing Methodology using the cyclomatic Complexity Metric", NIST Special Publication 500-235, August 1996.

[21] T.J. McCabe, "A Complexity Measure", IEEE Trans. Software Testing Engineering, N°2, December 1976, pp. 308-320.

[22] J. Poole, "A Method to Determine a Basis Set of Paths to Perform Program Testing", NISTIR 5737, November 1995.

[23] F. Ferrandi, M. Rendine, and D. Sciuto, "Functional Verification for SystemC Descriptions Using Constraint Solving," in Design Automation and Test in Europe (DATE'02) (C. D. Kloos and J. da Franca, eds.), (Paris, France), pp. 744–751, 4-8 March 2002.

[24] F. Baray, P. Codognet, D. Diaz and H. Michel. ``Validation of Functional Processor Descriptions by Test Generation''. International Conference on Using Hardware Design and Verification Languages (DVCon), San Jose, USA, 2003.

[25] ILOG Solver 6.0 User's Manual, October 2003.

[26] L. Capocchi, F. Bernardi, D. Federici, P. Bisgambiglia, A DEVS-based Modeling and Behavioral Fault Simulator for RTL-level digital circuits, SCSC 2004, San-Jose, California, USA.

[27] C. Paoli, M-L. Nivet, J-F. Santucci et T. Campana, Path-Oriented Test Data Generation of Behavioral VHDL Description, IEEE International Workshop on Electronic Design, Test & Applications(DELTA'02), 29-31 January 2002, Christchurch, New Zealand, pp. 382-386.

[28] C. Paoli, M-L. Nivet et J-F. Santucci, "Use of constraint solving in order to generate test vectors for behavioral validation", Proc. IEEE International High Level Design Validation and Test Workshop, November 2000, Berkeley, Californie, USA, pp. 15-20.

[29] 1999 International Test Conference benchmarks. Benchmarks from Politecnico di Torino. http://www.cad.polito.it/tools/poli.itc.tar.gz.