Automatic Test Bench Generation for Validation of RT-level Descriptions: an Industrial Experience

F. Corno, M. Sonza Reorda, G. Squillero

Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy
http://www.cad.polito.it/

A. Manzone, A. Pincetti

Centro Ricerche FIAT
Sistemi Elettronici, Progettazione HW
Orbassano (TO), Italy
http://www.crf.it/

Abstract

In current microprocessors and systems, an increasingly high silicon portion is derived through automatic synthesis, with designers working exclusively at the RT-level, and design productivity is greatly enhanced. However, in the new design flow, validation still remains a challenge: while new technologies based on formal verification are only marginally accepted, standard techniques based on simulation are beginning to fall behind the increased circuit complexity. This paper proposes a new approach to simulation-based validation, in which a Genetic Algorithm helps the designer in generating useful input sequences to be included in the test bench. The technique has been applied to an industrial circuit, showing that the quality of the validation process is increased.

1 Introduction

In the past years the ASIC design flow experienced radical changes, and due to the maturity of automatic logic synthesis tools most of the design activity is now performed at the RT-level, or higher. One of the main advantages of the new flow is the greater designer productivity, coming basically from the reduced size and higher readability of high level descriptions.

One important step of the new design flow still consists of *design validation* at the RT-level, i.e., the verification that the design is correct before starting logic synthesis and implementation. Although many techniques have been proposed in the past (e.g., static checks, formal verification [HuCh98], mutation testing [AHRo98]), none has gained enough popularity to compete with the current practice of *validation by simulation*. Designers (or verification engineers) typically resort to extensive simulation of each design unit, and of the complete system, in order to gain confidence over its correctness.

This situation is far from ideal, and designers need to face many difficulties. Simulation technology is nowadays quite effective for synthesized circuits, but when it comes to mixed-signal circuits, or to complete systems composed of several boards, or to circuits containing embedded cores or large memories, simulation becomes very expensive (in terms of required CPU times), without providing the needed performance and versatility.

Even if we restrict our attention to digital synthesized circuits, the fundamental issue of measuring the test bench *quality* is often unanswered. Many metrics have been proposed to evaluate the thoroughness of a given set of input stimuli, often adopted from the software testing domain [Beiz90], ranging from statement or branch coverage, state coverage (for finite state machine controllers), condition coverage (for complex conditionals), to the more complex path coverage. Many variants have been developed, mainly to cater for observability [DKGe96] and for the inherent parallelism of hardware descriptions [TAZa99], that are not taken into account by standard metrics. Since no well established metric is yet widely accepted for validation, some authors also propose to measure the quality of validation patterns with the stuck-at fault coverage.

Several products (normally integrated into existing simulation environments) are now available that provide the user with the possibility of evaluating the coverage of given input stimuli with respect to a selected metric. Designers can therefore pinpoint the parts of their design that are poorly tested, and develop new patterns specifically addressing them. Currently, this is a very time consuming and difficult task, since all the details of the design must be understood for generating suitable input sequences. The right trade-off between designer's time and validation accuracy is often difficult to find, and this often results in under-verified circuits. Moreover, in the generation of test vectors the designer may be "biased" by his knowledge of the desired system or module behavior, so that he often fails in identifying input sequences really able to activate possible critical points in the description.

When faced with this problem, the CAD research community traditionally invested in *formal verification* [GDNe91] [HuCh98], in the hope that circuits can be *proven* correct by mathematical means. Although formal verification tools give good results on some domains, they still have too many limitations or they require too much expertise to be used as a mainstream validation tool. Designers are left waiting for the perfect formal verification system, while few or no innovative tools help them with simulation-based validation.

The main goal of this paper is to propose an automatic input pattern generation tool able to assist designers in the generation of a test bench for difficult parts of the design. The approach we propose is suitable for simulation-based validation environments, and aims at integrating, rather than replacing, current manual simulation practices.

While no metric is yet widely accepted by validation teams, we aimed at evaluating the effectiveness of our approach using some pre-defined metric. The algorithm is quite easily adapted to different metrics, but for the sake of the experiments we adopted branch coverage as a reference. We developed a prototypical system for generating test patterns based on branch coverage, applicable to synthesizable VHDL descriptions. We aim at addressing moderately sized circuits, that usually can not be handled by formal approaches, and at working directly on the VHDL description, without requiring any transformation nor imposing syntax limitations.

The technique is based on a Genetic Algorithm, interacting with a VHDL simulator, that automatically derives an input sequence able to execute a given statement, or branch, in the RTL code. Whenever the test bench quality, as measured by one of the proposed metrics, is too low, our tool can be used to generate test patterns that are able to stimulate the parts of the design that are responsible for the low metric. The designer must manually analyze only those parts of the description that the tool failed to cover. Experimental results show that only a small fraction of “difficult” statements remain uncovered, and that many of them, upon closer inspection, indeed contain design errors or redundancies.

In previous works we already applied Genetic Algorithms to generate sequences for performing approximate equivalence verification between gate-level [CSSq98] or RT-level [CSSq99] descriptions. The approach presented in this paper is radically different, since it aims at increasing an independent quantitative metric, rather than finding at least a difference in a couple of circuits.

The approach has been evaluated by applying it during the design and validation of a circuit in Centro Ricerche FIAT, the research and development center of a leading automotive industry. The tool was seamlessly integrated in the design flow, without requiring circuit modifications or remodeling steps. Experimental results show that the manually derived validation suite did not adequately cover some parts of the design, and that new sequences have been generated by the tool to increase the overall coverage. Some portions still remained uncovered, and required a manual analysis that identified some design redundancies.

The ability of generating RT-level test patterns that reach some specific goal makes a wide range of applications possible. Besides automatic test bench generation, that is described in this paper, and approximate equivalence checking [CSSq99], an interesting application is generation of test patterns for production testing (ATPG). Test pattern generation requires input sequences that are able to test faults in the netlist, and this concept can be approximated at the RT-level by *controlling* statements and branches (i.e., executing them, as in design validation) and by *observing* them (i.e., propagating their effects to at least some primary output). A different version of the algorithm presented in this paper has been developed for test generation [CSSq00] and is shown to reach a stuck-at Fault Coverage on the synthesized gate-level description which is comparable with the one obtained by a gate-level ATPG.

Section 2 gives an overview over the proposed approach for test bench generation, while experimental results on the industrial circuit are presented in Section 3. Section 4 concludes the paper.

2 RT-level Test Bench Generation

The goal of test bench generation is to develop a set of input sequences that attain the maximum value of a predefined validation metric.

2.1 Adopted Metric

Most available tools grade input patterns according to metrics derived from software testing [Beiz90]: statement coverage and branch coverage are the most widely known, but state/transition coverage (reaching all the states/transitions of a controller) and condition coverage (controlling all clauses of complex conditionals) are also used in hardware validation. Path coverage, although often advocated as the most precise one, is seldom used due to its complexity, and because it loses meaningfulness when multiple execution threads run concurrently in parallel processes. Some recent work extends those metrics to take also into account observability [DGKe96] and the structure of arithmetic units [TAZa99]. Those extensions are essential when the sequences have to be used as test patterns to cover stuck-at faults, but for validation they have lower importance since internal values are available.

The metric we adopt in this paper is branch coverage, although the tool can be easily adapted to more sophisticated measures. Also, since synthesizable VHDL is a structured language, complete statement coverage implies complete branch coverage, and the tool takes advantage of this simplification.

2.2 Overall Approach

The adopted approach is an evolution of the one presented in [CPSo97], where a Genetic Algorithm uses a simulator to measure the effectiveness of the sequences it generates. Instead of trying to justify values across behavioral statements, that would require solving Boolean and arithmetic constraints [FADe99], thanks to the nature of Genetic Algorithms we just need to simulate some sequences and analyze the propagation of values. Each sequence is therefore associated with the value returned by a *fitness function*, that measures how much it is able to enhance the value of the validation metric, and the Genetic Algorithm evolves and recombines sequences to increase their fitness.

The fitness function needs to be carefully defined, and accurately computed. In particular, the fitness function can *not* be just the value of the validation metric: it must also contain some terms that indicate how to *increase* the covered branches, not just to *count* the already covered ones. In a sense, the fitness function includes a dominant term, that measures the accomplished tasks (covered branches), and secondary terms, that describe sub-objectives to be met in order to cover new branches.

The computation of such function is accomplished by analyzing the simulation trace of the sequence, and by properly weighting the executed assignments, statements, and branches according to the target statements. In the implementation, to avoid arbitrary limitations in the VHDL syntax, simulation is delegated to a commercial simulator that runs an *instrumented* version of the VHDL code and records the simulation trace in the transcript file. Such trace is then interpreted according to control- and data-dependencies, that are extracted from a static analysis of the design description. Figure 1 shows a simplified view of the overall system architecture.

2.3 VHDL Analysis

The goal of the algorithm is to achieve complete coverage, but for efficiency reasons we do not consider each statement separately, and we group them into *basic blocks* [ASUl86]: a basic block is a set of VHDL statements that are guaranteed to be executed sequentially, i.e., they reside inside a process and do not contain any intermediate entry point nor any control statement (`if`, `case`, …). All the operations required for code instrumentation, dependency analysis, branch coverage evaluation, and fitness function computation are performed at the level of basic blocks.

Since the Genetic Algorithm exploits the knowledge about data and control dependencies, we need to extract that information from the VHDL code: for this reason, we build a *database* (Fig. 2) containing a simplified structure and semantics of the design. The database is structured as follows:

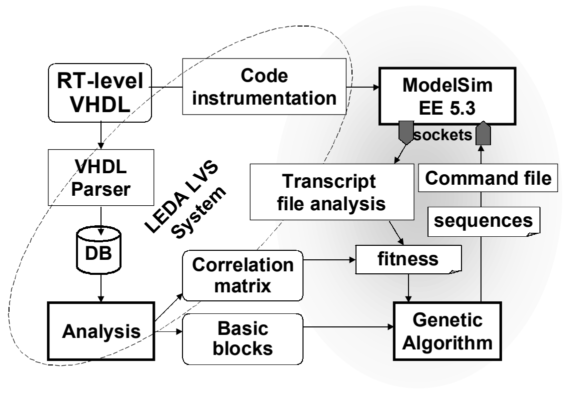

Figure 1: System architecture

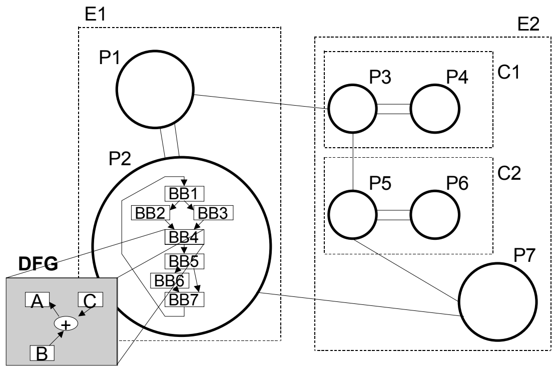

Figure 2: Abstract representation of RT-level designs

- The hierarchy of component instantiations inside different entities is *flattened* (C1 and C2 in the figure). A dictionary of *signal equivalencies* is also built, that allow us to uniquely identify signals that span multiple hierarchical levels.
- All VHDL *processes* occurring in the flattened circuit are given a unique identifier (P*i* in the figure). This operation also converts standalone concurrent statements into their equivalent process. The design is thus represented as a *network of processes interconnected by signals*.
- Each process is analyzed to define its *interface*, in terms of signals that it reads and writes.

- •The sequential part of each process is analyzed, its *control flow graph* (CFG) is extracted, and statements are grouped in *basic blocks* (BBs). The control structure of the process is described as a control flow of basic blocks (the figure reports the CFG for process P2).
- •A *dependency matrix* between basic blocks is computed, by assigning a probability that a basic block will be executed, given that another block has just been executed. These correlation probabilities take into account the branching and looping nature of the control flow.
- •Each basic block is entered, and the *data flow graph* (DFG) of the operations that occur inside each basic block is extracted. Since a basic block consists of multiple statements and/or conditions, multiple dependencies are associated to a single block. Fig. 2 shows the DFG for the basic block BB4 of process P2.

2.4 Genetic Algorithm

The Genetic Algorithm (GA) is based on encoding potential test sequences as variable length bit matrices. A number of such sequences are randomly generated and constitute an initial *population*: the goal of the GA is to evolve this population to increase its *fitness* value. The fitness function measures the closeness of a sequence to the goal; currently, this function assumes different forms in the two phases in which the algorithm is organized.

In a *first phase*, the goal is to find a single sequence that, when applied from the reset state, activates the highest number of branches. In this phase, the fitness function is simply the number of branches that have been traversed at least once.

In the *second phase*, each yet uncovered branch is considered separately as a *target*, and a genetic experiment is run whose goal is to execute it. For each experiment, the fitness function computes the closeness of the current sequence to reaching the target, measured as the weighted average of the execution counts of the basic blocks in the input cone (taking into account both control and data dependencies, thus potentially spanning several processes) of the target. The adopted weights take into account the probabilities of conditional execution that were statically computed in the database.

3 Experimental results

The goal of the experimental evaluation was to apply the proposed validation methodology to an industrial design flow, in particular by selecting a circuit as a case study.

The implementation consists of about 4,700 lines of C code for VHDL code analysis and instrumentation, linked to the LEDA LPI interface [LEDA95], and of 2,700 lines of C code for the Genetic Algorithm and the interface to the simulator. All experiments were run on a Sun Ultra 5 running at 333 MHz with 256MB of memory.

We applied the test bench generation procedure to an industrial circuit designed at Centro Ricerche FIAT (CRF).

The circuit developed at CRF is essentially a large Finite State Machine which takes care of driving the control signals used to actuate electronic injectors in diesel engines. It is capable of driving 6 injectors separately, giving the possibility to an external microprocessor to program the actuator timings by writing some parameters into an internal 2-port RAM through a 16 bit data bus. The prototypal version of this circuit has been developed using a Field Programmable Gate Array (Xilinx XC4028EX) while the production version is being developed on an ASIC.

This circuit is completely digital and described in synthesizable VHDL, and it contains memories, internal 3-state busses, and a microprocessor interface. Its main characteristics are summarized in Tab. 1.

Parameter	Value
VHDL lines	10,013
VHDL entities	118
VHDL process declarations	116
VHDL process instances	290
Basic blocks	1,465
Primary Inputs (ports/bits)	27 / 78
Primary Outputs (ports/bits)	21 / 47
Flip-Flops	452
Equivalent gates	~12,000

Table 1: Industrial circuit characteristics

The VHDL description was taken without any modification, and was analyzed to build the database and instrumented. This proves that the tool can be inserted very easily into an existing design flow, and requires only a marginal design effort. The Genetic Algorithm was run for about 150 CPU hours, heavily dominated by simulation time, during which it generated 5,219 vectors, achieving a coverage of 77.13%. As a comparison, the test bench developed by the designer, that consisted of about 400 vectors, reached a coverage of 63.17%, only. This proves that, with a negligible designer effort and an acceptable CPU time, test bench quality (and therefore validation effectiveness) is significantly improved. To prove that this result is due to the Genetic Algorithm, we compared the attained coverage with one coming from an equal number of completely random patterns: in this case the coverage is only 59.31%, showing that a clever optimization algorithm is effective and necessary over pseudo-random simulation.

These results prove that the test bench generation algorithm is able to improve the quality of validation without any effort from the designers, and with an acceptable CPU time. The portions of the design that escaped both manual and automatic test benches are currently under analysis. Many branches correspond to redundant `else` or `default` branches for `if` or `case` statements that were inserted solely to prevent the synthesis tool from inferring sequential logic: these statements are unreachable by construction. The reset logic is another source of uncovered statements: since the asynchronous initialization sequence is applied before fitness computation starts, the reset instructions (that are indeed executed) are excluded from the count. Designers then analyzed the other uncovered VHDL branches, that are concentrated in few processes, and found some real design redundancies.

4 Conclusions

This paper presented a new approach to design validation based on automatic generation of a test bench. The approach resorts to a Genetic Algorithm that interacts with a simulator to generate new sequences able to increase the coverage of the test bench with respect to a predefined validation coverage metric.

The methodology has been applied to an industrial circuit currently in production at Centro Ricerche FIAT, and a preliminary version of the tool has been used to generate the test bench for it. Experimental results prove that the method is able to increase the quality of the validation process both over manual simulation and pseudo-random sequence generation. The tool results have also been useful as a feedback for better understanding the most difficult parts of the design from the validation point of view. Currently, the design team at CRF is analyzing the best way to accommodate the proposed methodology into their standard design flow.

5 References

[AHRo98] G. Al-Hayek, C. Robach: *From Design Validation to Hardware Testing: A Unified Approach*, JETTA: The Journal of Electronic Testing, Kluwer, No. 14, 1999, pp. 133-140

[ASUl86] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986

[Beiz90] B. Beizer, *Software Testing Techniques (2nd ed.)*, Van Nostrand Rheinold, New York, 1990

[CPSo97] F. Corno, P. Prinetto, M. Sonza Reorda: *Testability analysis and ATPG on behavioral RT-level VHDL*, Proc. IEEE International Test Conference, 1997, pp. 753-759

[CSSq98] F. Corno, M. Sonza Reorda, G. Squillero, *VEGA: A Verification Tool Based on Genetic Algorithms*, Intl. Conf. on Circuit Design, 1998, pp. 321-326

[CSSq99] F. Corno, M. Sonza Reorda, G. Squillero, *Simulation-Based Sequential Equivalence Checking of RTL VHDL*, ICECS'99: 6th IEEE Intl. Conf. on Electronics, Circuits and Systems, 1999

[CSSq00] F. Corno, M. Sonza Reorda, G. Squillero, *High-Level Observability for Effective High-Level ATPG*, to be presented at IEEE VLSI Test Symposium, 2000

[DGKe96] S. Devadas, A. Ghosh, K. Keutzer: *An Observability-Based Code Coverage Metric for Functional Simulation*, Proc. ICCAD'96

[FADe99] F. Fallah, P. Ashar, S. Devadas: *Simulation Vector Generation from HDL Descriptions for Observability-Enhanced Statement Coverage*, Proc. 36th DAC, New Orleans, 1999, pp. 666-671

[FDKe98] F. Fallah, S. Devadas, K. Keutzer: *OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification*, Proc. 35th DAC, 1998

[GDNe91] A. Ghosh, S. Devadas, A.R. Newton, *Sequential Logic Testing and Verification*, Kluwer, 1991

[HuCh98] S.-Y. Huang, K.-T. Cheng, *Formal Equivalence Checking and Design Debugging,* Kluwer, 1998

[LEDA95] LVS System User's Manual, LEDA Languages for Design Automation, Meylan (F), April 1995

[TAZa99] P.A. Thaker, V.D. Agrawal, M.E. Zaghloul: *Validation Vector Grade (VVG): A New Coverage Metric for Validation and Test*, VTS'99: IEEE VLSI Test Symposium, 1999, pp. 182-188