System-level Test Bench Generation in a Co-design Framework

M. Lajolo
NEC USA C&C Research Labs

L. Lavagno
Università di Udine - DIEGM

M. Rebaudengo, M. Sonza Reorda, M. Violante
Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy
http://www.cad.polito.it/

Abstract

Co-design tools represent an effective solution for reducing costs and shortening time-to-market, when System-on-Chip design is considered. In a top-down design flow, designers would greatly benefit from the availability of tools able to automatically generate test benches, which can be used during every design steps, from the system-level specification to the gate-level description. This would significantly increase the chance of identifying design bugs early in the design flow, thus reducing the costs and increasing the final product quality. The paper proposes an approach for integrating the ability to generate test benches into an existing co-design tool. Suitable metrics are proposed to guide the generation, and preliminary experimental results are reported, assessing the effectiveness of the proposed technique.

1. Introduction

In the last years, new technologies allowed to integrate entire systems on a single chip, called System-on-Chip (SOC). SOC products represent a real challenge not just from the manufacturing point of view, but even when design issues are concerned.

To cope with SOC designers requirements, researchers developed co-design environments, whose main characteristic is to allow the designer to quickly evaluate the costs and benefits of different architectures, including both hardware and software components. In these environments it is also possible to automatically synthesize both the hardware and software modules implementing the desired system behavior.

Validation of the behavioral specification provided by designers, as well as of the hardware and software modules generated by co-design environments is a key point. In fact, it is known that the sooner a bug is found in the design, the lower it is the cost for fixing it. Although many techniques have been proposed in the past (e.g., static checks, formal verification [1] [2], mutation testing [3]), none has proved to be so effective to compete with the current practice of validation by simulation. Designers typically resort to extensive simulation of each design unit, and of the complete system, in order to gain confidence over its correctness.

In order to evaluate the goodness of the adopted input stimuli, several metrics have been proposed, which often come from the software testing domain [4], ranging from statement or branch coverage, state coverage (for finite state machine controllers), condition coverage (for complex conditionals), transition pair coverage (for protocols), to path coverage. Variants have also been developed, mainly to cater for observability [5] and for the inherent parallelism of hardware descriptions [6], that are not taken into account by standard metrics.

Several products (e.g. 0-in [7], SpecmanElite [8]), often integrated into existing simulation environments, now 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 exercised by the current stimuli, 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 systems. 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 [1] [2], 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.

The problem of generating a "smart" test bench for simulation-based design validation has been addressed, for example, in [9], where a Genetic Algorithm computes input sequences that cover the statements of register transfer level (RTL) VHDL models. Approaches targeting hardware systems, such as [5] [6] [7] [8] [9], need significant changes in order to be adapted to system-level design approaches that mix hardware and software components. Moreover, a top-down system level design methodology implies two requirements on a test-bench generation methodology:

- it is mandatory to validate the system as early as possible, if possible starting from the functional level, thus guaranteeing the correctness of designs before proceeding to the following synthesis steps;
- validation vectors should be reusable, i.e., the same test bench should be used during design validation at every level of abstraction. This allows to use the developed test sequences, for example, to verify the correctness of implementation steps like software synthesis or communication refinement, by comparing the input/output sequences of the two abstraction levels [10].

To satisfy these constraints, we propose to directly integrate test bench generation into a co-design environment. We can thus abstract system behaviors from their architectures; therefore, hardware and software components are treated in an uniform way; moreover, architecture-related annotations (such as delay models) can be taken into account.

The main goal of this paper is to propose an automated approach to assist designers in the generation of a test bench for system-level design. The approach that we propose is suitable for simulation-based validation environments, and aims at integrating, rather than replacing, current manual simulation practices.

A prototype of the proposed automatic input pattern generation technique has been implemented using the POLIS [11] co-design tool. It is based on a evolutionary algorithm, that automatically derives an input sequence able to exercise as much as possible of the specification, by interacting with a simulator executing a specification of the system under analysis. It can be easily adapted to different metrics, but for our first experiments we adopted edge coverage and statement coverage as a reference: i.e., the target of our algorithm is to traverse all the edges/statements of each module in the system specification. Edge coverage is intended to address control-oriented modules where control operations (e.g., if-then-else statements) predominates over computation. Conversely, statement coverage is more suitable for data-oriented modules.

The remainder of the paper is organized as follows. Section 2 presents our test bench generation approach, while Section 3 reports some preliminary results. Finally, Section 4 draws some conclusions and outlines future works.

2. 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.

Most available tools grade input patterns according to metrics derived from software testing: 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 metric, is seldom used due to its complexity, and because it is harder to apply meaningfully when multiple execution threads run concurrently in parallel processes. Some recent works extend those metrics to take also into account observability [5] and the structure of arithmetic units [6].

The metrics that we adopt in this paper are edge coverage and statement coverage in the individual graphs generated by POLIS for each system component (see Section 2.2), although the tool can be easily adapted to more sophisticated measures.

2.1. System representation: network of CFSMs

In POLIS the system is represented as a network of interacting Codesign Finite State Machines. CFSMs extend Finite State Machines with arithmetic computations without side effects on each transition edge. The communication edges between CFSMs are events, which may or may not carry values. A CFSM can execute a transition only when an input event has *occurred.*

A CFSM network operates in a *Globally Asynchronous Locally Synchronous* fashion, where each CFSM has its own *clock*, modeling the fact that different resources (e.g., HW or SW) can operate at widely different speeds. CFSMs communicate via non-blocking depth-one buffers. Initially there is no relation between local clocks and physical time, that gets defined later by a process called *architectural mapping*.

This involves allocating individual CFSMs to computation resources and assigning a scheduling policy to shared resources. CFSMs implemented in hardware have local clocks that coincide with the hardware clocking. CFSMs implemented in software have local clocks with a variable period, that depends both on the execution delay of the code implementing each transition and on the chosen scheduling policy (e.g., allowing task preemption).

As will be described in the following Sections, our approach requires to perform several simulations as fast as possible. Due to this requirements, we synthesize all the CFSMs in the system as software tasks, and execute them as native object code on the machine where simulations are performed. Note however that our approach is general because it is able to capture the reactive behavior of a network of CFSMs in which functionality and timing are interdependent and it can generate different test vectors for different hardware/software partitioning schemes [10].

2.2. Simulation model

Figure 1 shows an example of the Control Flow Graph representation that we use for software synthesis, called S-Graph.

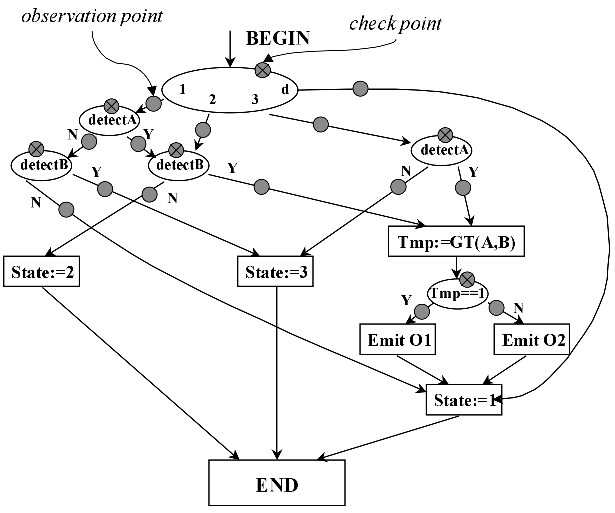

Figure 1: An *S-Graph example*

An S-Graph has a straightforward and efficient implementation as sequential code on a processor. In the C code that POLIS is able to generate from the S-Graph representation, each statement is almost in a 1-to-1 correspondence with a node in the S-Graph.

In order to check for the coverage of each edge/statement in a CFSM and to gather information during simulation of an input sequence, we instrument the C simulation model by inserting:

1. *observation points* associated to the edges in the S-Graph. An observation point is a fragment of C code that, whenever the edge is traversed, sets the bit associated with the observation point in a path identifier variable;
2. *trace points* associated to each statement in the S-Graph. A trace point is a fragment of C code that records the number of times the statement has been executed;
3. *check points* associated to each test in the S-Graph. A check point is a fragment of C code that records the number of times the test has been executed.

During simulations, we use the first two points to evaluate edge and statement coverage, and we use the last one to direct the search towards repeated traversal of test nodes that have still uncovered outgoing branches.

Note that we are evaluating our coverage on a *software-oriented* representation of the transition function of the CFSM. This means that our coverage measure has *physical meaning* mostly for a software implementation (e.g., in order to use the generated test-bench to test for faults in the program memory and in the processor that executes the code). However, it can also be used to help identify specification and implementation errors at the *uncommitted, functional level* as well as for hardware blocks [10].

2.3. Adopted evolutionary algorithm

The Selfish Gene algorithm (SG) is an evolutionary optimization algorithm based on a recent interpretation of the Darwinian theory. It evolves a population of individuals seeking for the fittest one. In the selfish gene biological theory, population itself can be simply seen as a pool of genes where the number of individuals, and their specific identity, are not of interest. Therefore, differently from other evolutionary algorithms, the SG resorts to a statistical characterization of a population composed by an *infinite number of individuals*, by representing and evolving some statistical parameters only. Evolution proceeds in discrete steps: individuals are extracted from the population, collated in tournaments and winner offspring is allowed to spread back into the population.

An individual is identified by the list of its genes. The whole list of genes is called genome and a position in the genome is termed locus. Each locus can be occupied by different genes. All these candidates are called the gene alleles. In the context of an optimization problem, looking for the fittest individual corresponds to determining the best set of genes according to the function to be optimized. Implementation details about the SG algorithm are available in [12].

2.4. Overall approach

In our approach, the SG algorithm evolves a populations of input sequences. In other words, it evolves a set of test benches looking for the fittest one.

Let L be the user-specified minimum test length and *NPI* the numbers of system input events, an individual $S = \{v_0, v_1, ..., v_{L-1}\}$ in the population is a stream of L input vectors. Each $v_t \in S$ is a vector of *NPI* integers, one for each input event. In our simulation model, all the events $e \in v_t$ will be concurrently placed at the system inputs, while the events belonging to v_{t+1} will be placed at the system inputs only when all the events belonging to v_t have been processed ("fundamental mode" operation).

Given an input event $e \in v_t$, the integer $n \geq 0$ that we associate to *e* has a different meaning depending on the type of the event *e*:

1. if $n = 0$, the event *e* is not placed on the system inputs during application of vector v_t;
2. if $n > 0$ and *e* is an event without value, *n* represents the number of event *e* that will be serially placed on the system inputs starting from v_t;
3. if $n > 0$ and *e* is an event with value, it will be placed on the system input with value *n*.

SG computes new individuals by modifying the integer *n* associated to each input event. Thanks to this coding, both the input values and test length become dimensions of the search space explored by the SG[1].

The proposed approach requires to simulate each input sequences and analyze its effects on the system. We associate to each sequence the value returned by a fitness function, that measures how much the sequence is able to enhance the value of the adopted validation metric. The task of the SG algorithm is thus to evolve and modify the population of test benches in order to maximize the fitness function.

The fitness function that we use in this paper is defined as follows:

$$f(S) = C_1 \cdot \sum_{i=0}^{N} OP_i + C_2 \cdot \sum_{j=0}^{M} \sum_{i=0}^{Nj} (1 - OP_i) \cdot NT_j \qquad (1)$$

Where:

1. *S* is the input sequence to be evaluated;
2. *N* is the number of observation points;
3. *M* is the number of check points;
4. N_j is the number of observation/trace points associated to checkpoint *j*;
5. OP_i is equal to 1 if the edge has been traversed, 0 otherwise;
6. NT_j is the number of times the check point associated to the test *j* has been executed during the simulation of the input sequence *S*;
7. C_1 and C_2 are two constants.

The first part of the above fitness function measures how many edges/statements the sequence *S* traverses. The second part tends to favor sequences that execute those tests whose outgoing branches have not yet been covered. In order to preserve the already covered edges while trying to cover new ones the first part must dominate over the second one (in the experiments we used C_1=1,000 and C_2=10).

[1] The different encodings are used to speed up the test pattern generation, by biasing it towards relatively long sequences of value-less events, that are generally used to drive counters.

3. Experimental results

We implemented a prototype of the proposed algorithm, called *Automatic Test Bench Generator* (ATBG), using the SG library developed at Politecnico di Torino.

The purpose of these experiments was to assess the feasibility of the proposed approach, therefore we ran some experiments on a set of benchmarks taken from a reasonably complex case study in the automotive domain: a car dashboard controller. All the results reported in table 1 have been gathered on a Sun UltraSparc 5 running at 333 MHz equipped with 384Mb of RAM.

Table 1 reports, for the chosen benchmarks, the CFSMs that constitute the benchmark, the number of edges/statements and the number of edges/statements that ATBG was able to cover when the maximum test length is set to 1,000. For the sake of comparison, we also report the number of edges/statements that 10,000 random vectors cover; the CPU time, not shown here, is less then 10 s for each experiment, including both the time spent by SG to evolve 100 generations and the relative simulation time.

Figure 2 shows an example of the effectiveness of ATBG vectors versus random ones on the BELT_CONTROLLER.

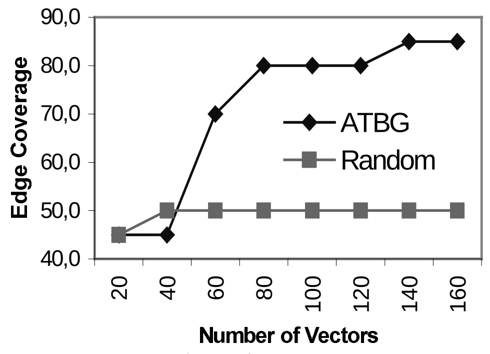

Figure 2: *ATBG and Random coverage over time*

In Table 2 we compare the edge/statement coverage of 10,000 ATBG-generated vectors with that of 10,000 random vectors and with that of 10,000 hand-written functional vectors. The functional test bench has been provided by a designer with an in-depth knowledge of the dashboard benchmark. Table 2 suggests that, even in its preliminary version, ATBG is able to automatically provide vectors that are comparable (5% better when edge covered is concerned, 2.5% better in the case of statement coverage) to those an experienced designer provides.

We are currently analyzing the behavior of ATBG in order to improve its coverage results. We have currently identified two sources of problems that a smarter fitness function should address:

- embedded CFSMs: in large models several CFSMs are deeply embedded in the design hierarchy; they receive events other CFSMs produce and they are poorly controllable from the system inputs.
- counter-like CFSMs: the fitness function has to be improved to deal with embedded CFSMs implementing counter-like behaviors that count for very long period of time (such as TIMER in table 2). In this case, the algorithm must generate the correct sequence of events enabling the counter and let it enabled for a suitable period of time.

4. Conclusions and future work

This paper presented an approach to automatic test bench generation intended for simulation-based validation of systems. The approach works in the very first phases of the design of a system, before the partitioning between hardware and software components is performed. The approach uses an evolutionary algorithm that interacts with a simulator to generate sequences able to reach a high coverage with respect to a predefined validation coverage metric.

A prototype has been developed, based on an existing co-design environment. Using this prototype, we assessed the feasibility of the proposed methodology on some simple benchmarks. Experimental results prove that the method is able to increase the quality of the validation process not only with respect to pseudo-random sequence generation but even when compared with sequence generated by experienced designers. As future work, we plan to exploit information that can be gathered on the S-Graph in order to identify and effectively address deeply embedded counter-like structures. Moreover, in order to overcome the existing limitations when addressing deeply embedded CFSMs, we are looking at ways of incorporating information about the structure of the network of CFSMs into the fitness function. Finally, some preliminary results show that the generated sequences can be of great help even in the phase of generating vectors for testing the hardware modules, since they embed a deep knowledge of the system behavior, and can be generated with very low CPU requirements.

5. References

[1] K. McMillan, *Symbolic Model Checking,* Kluwer, 1993

[2] R.P. Kurshan, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, Princeton Series in Computer Science, 1995

[3] 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

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

[5] S. Devadas, A. Ghosh, K. Keutzer: *An Observability-Based Code Coverage Metric for Functional Simulation*, Proc. ICCAD'96, pp. 418-425

[6] 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

[7] *http://www.0-In.com*

[8] *http://www.verisity.com/specmanelite.com*

[9] F. Corno, M. Sonza Reorda, G. Squillero, A. Manzone, A. Pincetti, " Automatic Test Bench Generation for Validation of RT-level Descriptions: an Industrial Experience ", to be presented at DATE'00

[10] H. Hsieh, A. Sangiovanni-Vincentelli, et al. "Synchronous equivalence for embedded systems: a tool for design exploration". Proc. ICCAD 99, pp. 505-509.

[11] F. Balarin et al., "Hardware-Software Co-design of Embedded Systems: The POLIS Approach", Kluwer Academic Publishers, 1997

[12] F. Corno, M. Sonza Reorda, G. Squillero, "The Selfish Gene Algorithm: a New Evolutionary Optimization Strategy", SAC'98: 13th Annual ACM Symposium on Applied Computing, 1998, pp. 349-355

Benchmark	CFSM	Edges [#]	Statements [#]	ATBG		Random	
				Edges [#]	Statements [#]	Edges [#]	Statements [#]
Belt Controller	BELT_CONTROLLER	20	31	17	28	10	20
	TIMER	13	25	11	22	10	17
Traffic Light Controller	CONTROLLER	28	66	26	63	10	28
	TIMER	8	13	7	12	7	12

Table 1: *ATBG vs. random vectors on small examples*

Benchmark	CFSM	Edges [#]	Statements [#]	Covered Edges [#]			Covered Statements [#]		
				ATBG	Random	Manual	ATBG	Random	Manual
Dashboard	BELT	17	25	11	11	7	16	16	12
	DISPLAY	12	73	7	3	7	53	39	53
	FRC	10	26	10	10	10	26	16	26
	FUEL	15	35	15	13	14	34	27	34
	ODOMETER	8	18	7	6	7	17	14	17
	TIMER	35	75	27	14	27	64	16	62
	SPEEDOMETER3	8	17	7	6	7	16	13	16
	SPEEDOMETER4	8	17	7	6	7	16	13	16
TOTAL		**113**	**286**	**91**	**69**	**86**	**242**	**154**	**236**

Table 2: *ATBG vs Random and Functional vectors on a large example*