Automatic Test Bench Generation for Simulation-based Validation

M. Lajolo  
NEC C&C Research Labs  
Princeton, NJ, USA  
lajolo@crl.nj.nec.com

L. Lavagno  
DIEGM  
Università di Udine  
Udine, ITALY  
lavagno@uniiud.it

M. Rebaudengo, M. Sonza Reorda, M. Violante  
DAI - Politecnico di Torino  
Torino, ITALY  
http://www.cad.polito.it

ABSTRACT

In current design practice synthesis tools play a key role, letting designers to concentrate on the specification of the system being designed by carrying out repetitive tasks such as architecture synthesis and technology mapping. However, in the new design flow, validation still remains a challenge: while new technologies based on formal verification are only marginally accepted for large designs, standard techniques based on simulation are beginning to fall behind the increased system complexity. This paper proposes an approach to simulation-based validation, in which an evolutionary algorithm computes useful input sequences to be included in the test bench. The feasibility of the proposed approach is assessed with a preliminary implementation of the proposed algorithm.

Keywords

Simulation-based validation, Genetic Algorithm.

1. INTRODUCTION

Thanks to the progress in automatic synthesis, system design flow is experiencing radical changes. The last years saw the advent of logic synthesis tools and today most of the design activity is performed at the RT-level. Now, system level design tools are appearing on the market, and a new step in the evolution of design practice, involving the introduction of more and more software in electronic systems, is on the way.

One important issue in the new design flow is design validation, i.e., the verification that the design is correct before submitting it to the following synthesis steps. Although many techniques have been proposed in the past (e.g., static checks, formal verification [1] [2], mutation testing [3]), none has gained enough popularity 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. Many metrics have been proposed to evaluate the thoroughness of a given set of input stimuli, often adopted 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. Many variants have 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 still at 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 us 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 [12]. To satisfy these constraints, we propose to integrate test bench generation in 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 as a reference: i.e., the target of our algorithm is to traverse all the transitions of each Codesign Finite State Machines in the system specification.

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 [4]: 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 work extends those metrics to take also into account observability [5] and the structure of arithmetic units [6].

The metric that we adopt in this paper is edge coverage in individual CFSMs, although the tool can be easily adapted to more sophisticated measures.

2.1 System representation

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 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 [12].

2.2 Simulation Model

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

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 in a CFSM and to gather information during simulation of an input sequence, we instrument the C simulation model by inserting:

- 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;
- check points associated to each node in the S-Graph containing a condition evaluation. 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 former to evaluate branch coverage, and we use the latter to direct the search towards repeated traversal of TEST nodes that have still uncovered 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 ROM). However, it can also be used to help identify specification and implementation errors at the uncommitted functional level as well as for hardware blocks [12].

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 [10].

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 $\mathcal{NI}$ the numbers of system input events, an individual $\mathcal{S} = \{v_0, v_1, \ldots, v_{L-1}\}$ in the population is a stream of $L$ input vectors. Each $v_i \in \mathcal{S}$ is a vector of $\mathcal{NI}$ integers, one for each input event. In our simulation model, all the events $e \in v_i$ will be concurrently placed at the system inputs, while the events belonging to $v_{i+1}$ will be placed at the system inputs only when all the events belonging to $v_i$ have been processed ("fundamental mode" operation).

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

- if $n = 0$, the event $e$ is not placed on the system inputs during application of vector $v_i$;
- 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 $v_i$;
- 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.

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:

$$f(S) = C_1 \cdot \sum_{i=0}^{N} \mathcal{OP}_j + C_2 \cdot \sum_{j=0}^{M} \sum_{i=0}^{N_i} (1 - \mathcal{OP}_j) \cdot \mathcal{NT}_j$$

Where:

- $S$ is the input sequence to be evaluated;
- $N$ is the number of observation points;
- $M$ is the number of check points;
- $N_i$ is the number of observation points associated to check point $j$;
- $\mathcal{OP}_j$ is equal to 1 if the edge has been traversed, 0 otherwise;
- $\mathcal{NT}_j$ is the number of times the check point associated to the test $j$ has been executed during the simulation of the input sequence $S$;
- $C_1$ and $C_2$ are two constants.

The first part of the above fitness function measures how many edges 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 $C_2=10$).

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 small benchmarks. All the results reported in table 1 have

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.
been gathered on a 120 MHz Pentium-based PC equipped with 64MB of RAM and running Linux RedHat 6.1.

Table 1 reports, for the chosen benchmarks, the CFSMs that constitute the benchmark, the number of edges, the input sequence length and the number of edges that ATBG was able to traverse. For the sake of comparison, we also report the number of edges that a 10,000 random vectors cover. The CPU time for all the experiments we performed is less than 60 s, 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 module.

![Figure 2: ATBG and Random coverage over time](image)

In Table 2 we compare the edge coverage of 10,000 ATBG-generated vectors with that of 10,000 random vectors and with that of 10,000 hand-written functional vectors. We plotted the same information in Figure 3. 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 (in this case, 4% better on average) 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 from 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 CFSMs implementing counter-like behaviors that count for very long period of time (such as TIMER in table 2).

4. CONCLUSIONS AND FUTURE WORK

This paper presented an approach to automatic test bench generation intended for simulation-based validation of systems. The approach uses an evolutionary algorithm that interacts with a simulator to generate sequences, in order to increase the coverage of the test bench with respect to a predefined validation coverage metric.

The methodology has been applied to some simple benchmarks in order to assess its feasibility. 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 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 CFSM into the fitness function.

5. REFERENCES


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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CFSM [#]</th>
<th>Edges [#]</th>
<th>ATBG [%]</th>
<th>Vectors [#]</th>
<th>Random [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belt Controller</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BELT_CONTROLLER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traffic Light Controller</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTROLLER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CFSM [#]</th>
<th>Edges [#]</th>
<th>Covered Edges [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>ATBG</td>
</tr>
<tr>
<td>Dashboard</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BELT</td>
<td>17</td>
<td>64.7</td>
<td>64.7</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>12</td>
<td>58.3</td>
<td>25.0</td>
</tr>
<tr>
<td>FRC</td>
<td>10</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>FUEL</td>
<td>15</td>
<td>100.0</td>
<td>86.7</td>
</tr>
<tr>
<td>ODOMETER</td>
<td>8</td>
<td>87.5</td>
<td>75.0</td>
</tr>
<tr>
<td>TIMER</td>
<td>35</td>
<td>77.1</td>
<td>40.0</td>
</tr>
<tr>
<td>SPEEDOMETER3</td>
<td>8</td>
<td>87.5</td>
<td>75.0</td>
</tr>
<tr>
<td>SPEEDOMETER4</td>
<td>8</td>
<td>87.5</td>
<td>75.5</td>
</tr>
<tr>
<td>AVERAGE</td>
<td></td>
<td></td>
<td>85.8</td>
</tr>
</tbody>
</table>

Figure 3: Comparing different approaches