Behavioral-level Test Vector Generation for System-on-Chip Designs

M. Lajolo
NEC USA C&C Research Lab

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 sequences, which can be reused during the following design steps, from the system-level specification to the gate-level description. This would significantly increase the chance of identifying testability problems 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 sequences into an existing co-design tool. Preliminary experimental results are reported, assessing the feasibility of the proposed approach.

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.

While the design practice is quickly moving toward higher levels of abstraction, test issues are still considered only when a detailed description of the design is available, typically at the gate-level for test sequence generation purposes and register transfer (RT)-level for design for testability structures insertion.

In the past years, intensive research efforts have been devoted to devise solutions tackling test sequence generation since the early design phases, mainly the RT-level and several approaches have been proposed. Most of them are usually able to generate test patterns of good quality, sometimes comparable or even better than gate-level ATPG tools. However, lacking of general applicability, these approaches are still not accepted by industries. The different approaches are based on different assumptions and on a wide spectrum of distinct algorithmic techniques. Some are based on extracting from a behavioral description the corresponding control machine [1] or the symbolic representation based on binary decision diagrams [2], while others also synthesize a structural description of the data path [3]. Some approaches rely on a direct examination of the HDL description [4], or exploit the knowledge of the gate-level implementation [5]. Some others combine static analysis with simulation [6].

Most of the cited approaches rely on high-level fault models for behavioral HDL descriptions that have been developed by the current practice of software testing [7] and extending them to cope with hardware descriptions. In this sense, the high-level fault model corresponds to a *metric* that measures the goodness of a given sequence of input vectors. Some of the difficulties that make de-

0-7695-0786-7/00 $10.00 © 2000 IEEE

veloping a good metric for hardware descriptions much more difficult than for software are: the large amount of concurrency that dramatically degrades the usefulness of the widely used path coverage or similar metrics; the combined presence of behavioral and structural description styles that prevents the use of techniques suitable for control-oriented or data-oriented circuits only; the complexity of timing schemes (multiple clock circuits are difficult to handle, as they were at the gate-level); the difficulty of modeling faulty behavior observation (since software metrics only consider reachability of conditions, that corresponds to fault controllability).

In this work, we propose an approach that targets the system-level of abstraction, allowing performing test sequence generation at the same level of abstraction the design is carried out. We propose an approach that addresses systems described at the behavioral-level and that computes test sequences attaining high fault coverage when applied to the corresponding gate-level model of the systems.

The approach is inspired to the ones described in [8] and [9]. The main contributions of this work are:

- the adoption of a high level fault model used to represent gate-level faults while at the behavioral-level;
- the development of a behavioral-level fault simulator to evaluate system behaviors in presence of faults.

Given a behavioral-level fault model and a fault simulator supporting it, we can evaluate the goodness of input stimuli from a testability point of view while reasoning at the behavioral-level, before a gate-level description is available. By exploiting this feature, we developed a behavioral-level test pattern generator (BL-TPG) and, to assess its feasibility, we compared the fault coverage of the sequences it produces while working at the behavioral-level with the one obtained by a commercial gate-level automatic test pattern generator.

Preliminary results show that input sequences computed at a higher level of abstraction could be exploited as test sequences at the lower level of abstraction. The results also show some weak points of the current implementation of our approach that demand further improvements.

The paper is organized as follows. Section 2 describes the adopted fault model, while Section 3 describes the behavioral-level fault simulator. Section 4 describes the BL-TPG tool. Finally, experimental results are presented in Section 5 and some conclusions are drawn in Section 6.

2. Fault model discussion

The effectiveness of test vectors is usually measured resorting to a gate-level description of the system under test and a fault simulation supporting one of the available gate-level fault models. One of the most popular fault models is the permanent single stuck-at one, thus we adopted it to measure how effective sequences are while at the gate-level.

When developing a behavioral-level test pattern generator we assume that, even if computed at a higher level of abstraction, the usefulness of the generated vectors is evaluated at the gate-level. Therefore, the behavioral-level test generator should adopt a behavioral-level fault model with a good correlation with respect to the adopted gate-level one. Moreover, the behavioral-level fault model should be consistent with the description of the system the design tool provides.

In our work we addressed the POLIS [10] co-design environment, and developed a fault model that mimics the permanent single stuck-at fault model while at the behavioral level.

In the following sub-sections we recall the system representation POLIS offers and then we present the behavioral fault model.

2.1. System representation: network of CFSMs

In POLIS the system is represented as a network of interacting Codesign Finite State Machines (CFSMs). CFSMs extend Finite State Machines with arithmetic computations without side effects on transition edges. 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).

2.2. Behavioral fault model

A behavioral fault model that approximates the stuck-at one can be defined by analyzing the synthesis rules exploited by POLIS to produce a gate-level model from a behavioral-level one. We reported in Table 1 the correspondence existing between the two levels.

Behavioral-level	Gate-level
Value on events	Word in registers
Presence/absence of events	Bit in flip-flops
Value on variables	Word in registers

Table 1: Correspondence between behavioral-level and gate-level

Given the correspondence reported in Table 1, we adopted permanent single stuck-at in events and variables of behavioral descriptions as the behavioral-level fault model. Figure 1 represents the relation between the set of gate-level (GL) faults and the behavioral-level (BL) one.

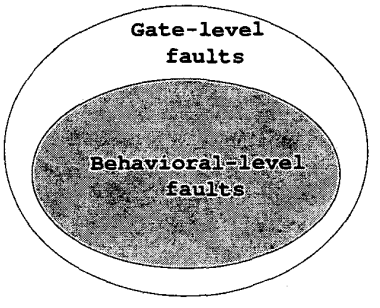

Figure 1: Gate- vs. Behavioral-level faults

We can observe that a one-to-one relation exists between GL and BL faults affecting memory elements. At the same time GL faults exist that cannot be represented at the BL, e.g., faults in the combinational logic. As a consequence, the set of BL faults is a subset of the GL one, where faults in the combinational logic are neglected.

While reasoning on the system behavior we adopted the BL fault model as a rough approximation of the GL model.

3. Behavioral-level fault simulator

The behavioral-level fault simulator has been developed in order to support the BL fault model previously described. It operates as a serial fault simulator and interacts with the simulation model POLIS provides.

Figure 2 shows an example of the representation that POLIS uses to model the behavior of each CFSM.

An S-Graph has a straightforward and efficient implementation as sequential code on a processor. Therefore, the behavior of a model composed of several interacting CFSMs can be simulated as a software program executed by a host workstation.

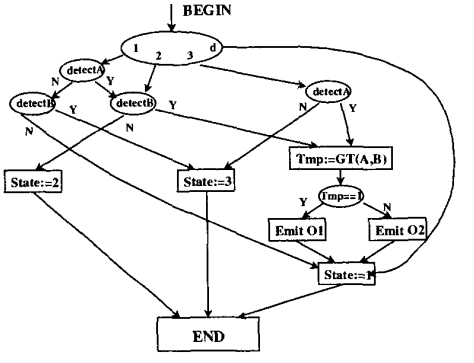

Figure 2: An S-Graph example

We exploited this approach and developed a fault simulator resorting to a source level debugger. In particular, given a bit B in the variable V of CFSM C as the current fault location, we set a breakpoint in every location of the source code for C where V is read. Then, during the simulation, every time a breakpoint is triggered, we modify the bit B in V according to the specified fault (either stuck-at-0 or stuck-at-1).

As far as response analysis is concerned, we dump on a file the output trace for the fault-free and the faulty behaviors, then after the simulation completion, we check if the two outputs match.

This approach, despite its simplicity, is very effective in allowing the simulation of faulty behavior. As a draw-

back, by operating in serial fashion, fault simulation could require significant CPU time for large models.

4. Test sequence generation

The purpose of the behavioral-level test pattern generation (BL-TPG) tool is to compute input sequences that maximize the fault coverage with respect to the BL fault model.

We adopted a heuristic approach that, starting from a given set of input vectors (e.g., manually developed input sequences), produces a sequence that maximizes the BL fault coverage (BL FC).

In our approach a test sequence is a set of vectors to be applied to the system. Each vector is a set of events that are concurrently applied to the system input at a given time. We coded the sequence as a matrix of bits, where `SEQUENCE_LENGTH` is the number of rows in the matrix and thus it represents the number of vectors to be applied on the system inputs. Conversely, `N_INPUTS` is the number of system inputs and thus the number of columns in the matrix. The number of bits used to represent an input event e is selected as follows:

- 1 bit if e is an input event without value
- $\log_2 n$ bits, where n is the number of different values associated to the event e, if e is an input event with value.

The algorithm we developed is described in Figure 3. For each fault f in the fault list (which comprises all the single permanent stuck-at faults in all the bit variables of every CFSM in the system) we executed a random mutation hill climber (RMHC) algorithm that randomly modifies an initial input sequence looking for a new one that detects f, i.e., a sequence for which the outputs of the faulty machine and the fault-free one differ. The RMHC exploits an evaluation function that measures how far a new sequence is from detecting the fault f. The function comprises two major terms:

1. the activity the sequence produces within each CFSM in the system, measured as the number of the statements coding the CFSM behavior that are executed by the sequence. To compute this information the techniques already described in [8] and [9] are exploited;
2. the number of CFSMs outputs in the faulty system (i.e., that system affected by the target fault) that hold a value not equal to the corresponding output in the fault-free system.

The first term is intended to traverse most of the system specification in order to excite the target fault, while the second term is used to reward those sequences that propagate a fault from the faulty site (a variable in a CFSM) to the CFSM outputs and possibly to the system outputs. The RMHC is stopped when the target fault is detected, or a maximum number of iterations has been performed.

In the current implementation of the algorithm the value of `SEQUENCE_LENGTH` is specified by the user.

```
void ATPG( sequence initial_S,
           faultlist F,
           int n_fault )
{
  int      i, j;
  sequence S;

  for( i = 0; i < n_fault; i++ )
  {
    S = initial_S;
    if( RMHC( F[i], S ) == DETECTED )
      save_sequence( S );
  }
}
```

Figure 3: Adopted test sequence generation algorithm

5. Experimental results

To assess the effectiveness of the proposed approach, we implemented a prototypical version of BL-TPG that amounts to 750 lines of C code (including the behavioral-level fault simulator). We run some experiments on three simple benchmarks whose number of CFSMs, number of statements coding the CFSMs behavior, number of primary inputs (PIs), number of primary outputs (POs), number of flip-flops (FFs) and number of gates are summarized in Table 2; all the CFSMs composing the benchmarks have been implemented as hardware modules. The benchmarks are the Traffic Light Controller (TLC), the seat belt controller (BELT) and a part of the dashboard (DASH) benchmark distributed with POLIS (we neglected the CFSMs containing trigonometric functions).

Name	CFSMs [#]	Statements [#]	PIs [#]	POs [#]	FFs [#]	Gates [#]
TLC	2	80	3	6	17	178
BELT	2	27	5	2	15	106
DASH	6	178	30	105	236	1,423

Table 2: Benchmarks characteristics

We run BL-TPG on a Sun UltraSparc 5/300 equipped with 256 MBytes of RAM to compute a set of test sequences for the behavioral descriptions of the adopted benchmarks. Then we fault simulated the attained vectors with the gate-level implementation of the benchmarks. We obtained the gate-level model of each benchmark by synthesizing its RTL-VHDL description (generated by POLIS) resorting to the Synopsys synthesis tool.

To evaluate the effectiveness of BL-TPG, we compared the fault coverage, *FC*, figures attained by the sequences it produces with the ones of random sequences of the same length, *Len*. Results in Table 3 shows that BL-TPG vectors are more effective than random generated ones.

Name	*BL-TPG*		*Random*	
	FC [%]	*Len [#]*	*FC [%]*	*Len [#]*
TLC	79.84	3,200	58.30	3,200
BELT	86.58	2,814	21.09	2,814
DASH	48.64	54,214	18.20	54,214

Table 3: *BL-TPG vs. random vectors*

Moreover, in Table 4 we compared the gate-level fault coverage figures BL-TPG attains with the ones of a commercial gate-level automatic test sequence generator (ATPG). Table 4 also reports the *CPU* required by test generation and fault simulation.

Name	*BL-TPG*			*Commercial ATPG*		
	FC [%]	*Len [#]*	*CPU [s]*	*FC [%]*	*Len [#]*	*CPU [s]*
TLC	79.84	3,200	915	84.38	567	3,651
BELT	86.58	2,814	870	86.58	755	3,661
DASH	48.64	54,214	3,618	64.63	19,662	18,035

Table 4: *BL-TPG vs. gate-level ATPG*

By observing this preliminary results the following considerations arise:

1. *fault coverage*: for the smallest benchmarks, BL-TPG produces results comparable to the ones a commercial ATPG produces; conversely BL-TPG falls short on the largest benchmark. This can be explained by considering that in the current implementation, our algorithm does not take into account the observability problem [6], and thus low fault coverage figures are attained;
2. *test length*: BL-TPG produces test sequences much longer than those a commercial ATPG produces. This is mainly due to the fact that the current implementation of BL-TPG does not support fault dropping;
3. *CPU time*: being based on a more abstract model than the gate-level one, BL-TPG requires less CPU time than the commercial ATPG. In particular, for the two benchmarks where similar fault coverage results are obtained, BL-TPG is 4 times faster than the adopted gate-level ATPG.

6. Conclusions

An approach to perform test sequence generation while at the behavioral-level has been proposed. The approach relies on a high-level fault model to map gate-level faults on the behavioral description of the system under test. A fault simulator supporting the behavioral-level fault model and a test generation tool have been developed and some experiments have been carried out to assess the feasibility of the proposed approach.

Experimental results show that when observability problems can be neglected, our approach is comparable to a commercial ATPG tool. The results are promising, but more work has to be done in order to consider observability during the test generation phase. Moreover, the BL fault model should be improved in order to consider also faults in the combinational logic.

7. References

[1] D. Moundanos, J. A. Abraham, Y. V. Hoskote, "A Unified Framework for Design Validation and Manifacturing Test," *Proceedings IEEE International Test Conference*, 1996, pp. 875-884

[2] F. Ferrandi, F. Fummi, D. Sciuto, "Implicit Test Generation for Behavioral VHDL Models," *Proceedings IEEE International. Test Conference*, 1998

[3] F. Fallah, P. Ashar, S. Devadas, "Simulation Vector Generation from HDL Descriptions for Observability-Enhanced Statement Coverage," *Proceedings 35th Design Automation Conference*, 1999, pp. 666-671

[4] S. Chiusano, F. Corno, P. Prinetto. "Exploiting Behavioral Information in Gate-Level ATPG," *JETTA: The Journal of Electronic Testing: Theory and Applications*, Kluwer Academic Publishers, No. 14, 1999, pp. 141-148

[5] E.M. Rudnick, R. Vietti, A. Ellis, F. Corno, P. Prinetto, M. Sonza Reorda, "Fast Sequential Circuit Test Generation Using High-Level and Gate-Level Techniques," *Proceedings IEEE European Design Automation and Test Conference*, 1998

[6] F. Corno, M. Sonza Reorda, G. Squillero, "High-Level Observability for Effective High-Level ATPG", *Proc. 18th IEEE VLSI Test Symposium*, pp. 411-416, 2000

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

[8] M. Lajolo, L. Lavagno, M. Rebaudengo, M. Sonza Reorda, M. Violante, "Automatic Test Bench Generation for Simulation-based Validation", *Proc. ACM/IEEE 8th International Workshop on Hardware/Software Codesign*, CODES, May 2000, pp. 136-140

[9] M. Lajolo, L. Lavagno, M. Rebaudengo, M. Sonza Reorda, M. Violante, "System-level Test Bench Generation in a Co-design Framework", *Proc. IEEE European Test Workshop 2000*

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