# Constrained-Random Verification for Synthesis: Tools and Results

Diana Bodean, Ghenadie Bodean, Olga Ghincul
*Technical University of Moldova*
*gbodean@gmail.com*

***Abstract* — This paper presents the tools for automation the synthesis of constrained-random generator for verification the synthesizable designs of microprocessors and microcontrollers. The structure of constrained-random generator is coded by a stochastic grammar that is defined using the elaborated tools. Various constrained-random parameters, inclusively simulation coverage, can be estimated thanks to correspondence between stochastic grammar and the Markov chain. The performed test experiments have showed that the apriori estimations and aposteriori test results are in good agreement..**

***Index Terms* — constrained-random, microcontroller, microprocessor, verification.**

## I. INTRODUCTION

Software tools for implementation new constrained-random VHDL features were presented on the web seminar that was held in spring 2009 [1]. "Are randomization codes synthesizable?" - was one of the main questions asked by the web participants. "No, they are not attempted to be what so ever. It's strictly a task of match thing" – was the answer.

In this paper the software tools, called RandGen, for data structures, inference rules and weight distribution description for constrained-random VHDL-templates generation are proposed and analyzed. The results of microprocessor and microcontroller constrained-random verification are also presented in this paper.

## II. PRELIMINARY

In this section is described the methodology of synthesis data structures for randomization. In our interpretation the term "constrained-random" is referred not only to randomization, but also for signing the *structural* (syntactical) as well as *algorithmic* (semantic) constrains.

When designing a constrained-random (CR) generator for *microprocessors* and *microcontroller* (MP) verification, the objective of test generation process is the synthesis of *test programs* with a specific syntax. In terms of formal linguistics, constraints on the structure of instructions as well as on its sequence in the test programs, are called *syntactic*. The syntax of instructions is formally determined in the specification of instructions set of unit under verification (UUV).

Syntax of the test program (TP) is generally determined by the coverage of UUV structural elements and data flows. From this point of view some MP instructions load the data in the memory's elements, others - process or/and unload the data from them [2]. Thus, the rule of definition of TP syntax can be formulated by the next paradigm:

"*data load* → *inherently data process* → *unload* (*and analysis*) *of results*". (1)

Rule (1) expresses also the *semantic* aspect of the objective of CR test program generation. Or, rule (1) reflects the stochastic generation process of test programs with data dependency and can be accepted as a link between functional (behavioral) model of the MP [2] and rules of synthesis the TP generator, proposed in [3, 4].

Except test program syntaxes when describing the generator behavior (test bench) it is required an "event-driven" simulation refereed to the unit under verification (UUV) reaction on events, such as internal and external *interrupts* and others. Such kind of constrains can be undoubtedly attributed to semantic ones. In addition, the *test-designer* must know the technology peculiarities of CR generations of test programs at constrains level as well as at the implementation level of TP generator.

So, when designing a TP constrained random generator, the test-designer must have, on one hand, advanced facilities that would provide high degree of freedom for describing various stochastic processes of generation, and on the other hand, the test-designer must have a full specification of the verified device in order to present possible syntactic, semantic and probabilistic (*stylistic*) constraints "imposed" on the process of generation of test programs. In terms of formal linguistics such kinds of constrains can be explicitly expressed by a *stochastic grammar*

## III. DEFINITION OF CONSTRAINED-RULES

A regular stochastic grammar is defined by a 5-tuple $G=(V_N, V_T, S, \Phi, P(\Phi))$ where $V_N$ and $V_T$ are sets of non terminal (syntactic class) and terminal symbols, respectively. $S$ is the starting symbol, $S \in V_N$. $\Phi$ is a finite set of rules of the form $\alpha \rightarrow \beta$, where $|\alpha| \leq |\beta|$, $\alpha \in V_N$, and $\beta$ has the form $aB$ or $B$, where $\alpha \in V_T$, and $\beta \in V_N$. $P(\Phi)$ is a distribution of probabilities (weights) supported by the rules of $\Phi$.

Let consider an example with non-stochastic grammar $G_1=(\{S, A, B, C\}, \{a, b\}, S, \Phi)$ and $\Phi$ is the set of rules (productions): $\Phi = \{S \rightarrow aB, B \rightarrow bC, C \rightarrow aC \mid aS \mid b\}$, where symbol | signs the alternative rule. The grammar $G_1$ generates a language $L$ starting from symbol $S$. A sentential form is any derivative of the unique symbol $S$. The language $L$ generated by the grammar $G$ is the set of all sentential forms whose symbols are terminal. In grammar

$G_1$ the sentence *abab* has the following derivation:

$$S \Rightarrow aB \Rightarrow abC \Rightarrow abaC \Rightarrow abab. \quad (2)$$

*Remarks*: in stochastic grammar the alternative rules will be labeled with transition probabilities (weights). For example, $C \xrightarrow{p_1} aC, C \xrightarrow{p_2} aS, C \xrightarrow{p_3} b$ are the weighted rules, where $p_1 + p_2 + p_3 = 1$.

In MP CR verification a sentence can specify an instruction as well as a test program. From the point of view of the paradigm (1) the objective of MP constrained random verification is generation of a *finite language* with maximum *coverage* of predefined *confidence* level. So, the *coverage* performance (i.e. the test quality) of CR-verification is estimated by stochastic language *length*.

As it is known that parsing the rules can be represented by a syntax tree [5]. A syntax tree is an important aid to understanding the syntax of a sentence. More generally, any sentential form, i.e. test program, has a syntax tree. The *average height* of syntax tree defines the length of CR generated TP.

Further, a scenario of the stochastic grammar definition will be presented on the example of PIC16C5x microcontroller (see specification on www.microchip.com). Each PIC16C5x instruction is a 12-bit word divided into an operation code OPCODE, which specifies the instruction type, and one or more operands, which specify the operation of the instruction.

Figure 1 shows four general formats that the instructions can have, where, for **byte-oriented** instructions, 'f' represents a file register designator and 'd' represents a destination designator, and for **bit-oriented** instructions, 'b' represents a bit field designator which selects the number of the bit affected by the operation, while 'f' represents the number of the file in which the bit is located. For **literal and control** operations, 'k' represents an 8- or 9-bit constant or literal value.

Formats shown in figure 1 define, in fact, the syntax of microprocessor instructions and in the same time specify the way of "assembly" the instruction: a random value (`std_logic_vector`) of constant, literal, or registers with a deterministic value of the code operation OPCODE.

In order to obtain an optimal (i.e. minimum) height of syntax trees the PIC16C5x instruction set must be divided in 7 subsets: Gr1, … Gr7, that forms the set of terminal symbols of generation grammar. In the RandGen terminologies the terminal symbols are called *lexemes*. The lexeme's names and decimal values are introduced in the

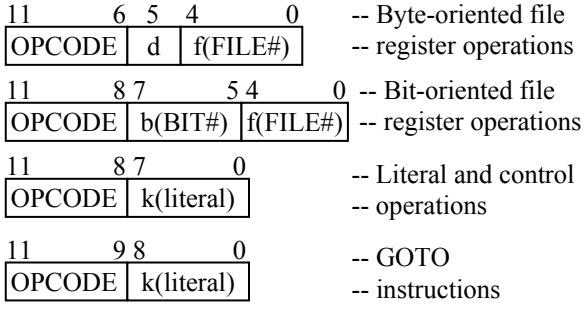


**Figure 1. PIC16C5x instructions general format**

lexemes list window (see figure 2), where Mask(·) specifies (by log.1) instruction bits which will be assigned by a random values. The set of lexemes is saved in the ROM

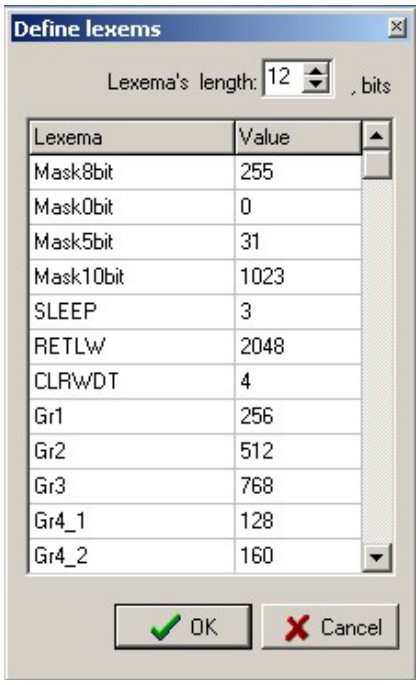


**Figure 2. Lexemes list window**

entity that is an inherited component of the CR template.

## IV. CONSTRAINED-RANDOM TEMPLATE

The RandGen-application can be viewed as a mediator between the test-designer and CAD-tools that facilitates and automates the constrained-random generator specifications. The rules of stochastic grammar, that will sequence lexemes, are introduced in the grid-window (Figure 3).

In the standard grammar format the table from Figure 3 without lexemes can be rewritten as:

$$A \xrightarrow{0,998} B,\ A \xrightarrow{0,002} B,\ B \to D,\ C \to J,$$
$$D \xrightarrow{0,3} E,\ D \xrightarrow{0,2} F,\ D \xrightarrow{0,05} G,$$
$$D \xrightarrow{0,25} H,\ D \xrightarrow{0,2} I,$$
$$E \xrightarrow{1/9} D, ..., E \xrightarrow{1/9} D,$$
$$F \xrightarrow{1/6} D, ..., F \xrightarrow{1/6} D,$$
$$G \to D,\ H \xrightarrow{1/7} D, ..., H \xrightarrow{1/7} D,$$
$$I \to A,\ J \to K,\ K \to A \quad (3)$$

It should be remarked that grammar (3) and weighed rules, e.g. D→ … in (3), are equivalent to the randomized case-statement in VHDL [1, 6] as:

```
variable RV: RandomPType;
. . .
--specifies just weight
RandVal:=RV.DistInt((207,138,23,161,125));
. . .
when D=>
  case RV.RandVal(1,2,3,4,5) is
    when 1=> Lexem< =Mask8bit; NextState<=E;
    when 2=> Lexem<= Mask5bit; NextState<=F;
    when 3=> Lexem<= Mask10bit;NextState<=G;
```

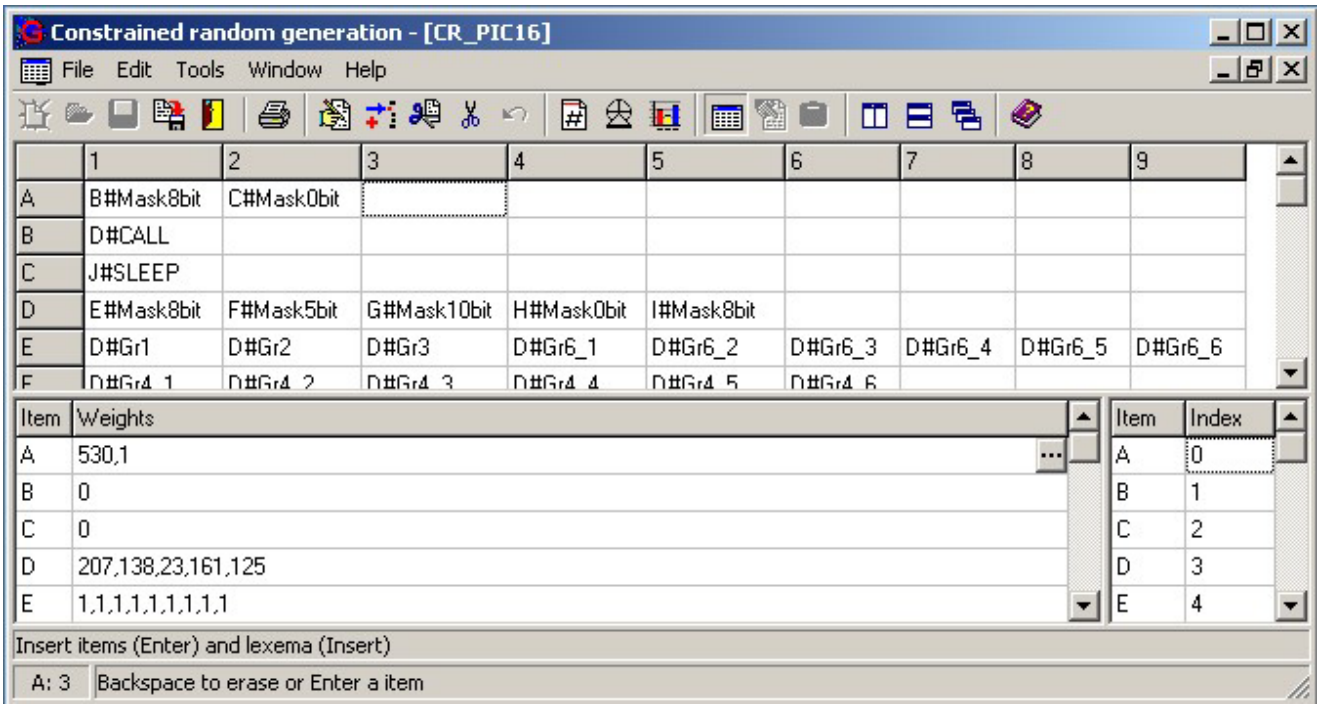


**Figure 3. RandGen User Interface**

```
    when 4=> Lexem<= Mask0bit; NextState<=H;
    when 5=> Lexem<= Masl8bit; NextState<=I;
  end case;
when E=> . . .
```

Distribution is introduced in weights window, shown in Figure 4. Moving the slider of trackers the test-designer performs weights definition. The weight Window also allows resizing the weights scale (right-top button in the Figure 4).

As the rules and weights are specified it remains only to click the button save to generate the corresponding VHDL-entities. An RTL-block diagram of instantiation the constrained random generator is shown in figure 5. The reset input (low level) initializes the RandGen generator that "operates" on rising edge of the clock signal **clk**. The output standard logic signal **RndBit** is a bit of the Counter Linear FeedBack Shift Register (LFSR), which is used as a random source. The low level of the **Ready** output indicates that the random code **RndCode** is generated. Simultaneously with **RndCode** a corresponding new **Item** and **Lexeme** is also generated.

From viewpoint of formal grammar, the variable **Item** is a non-terminal symbol. The set of items are predefined in the RandGen tools.

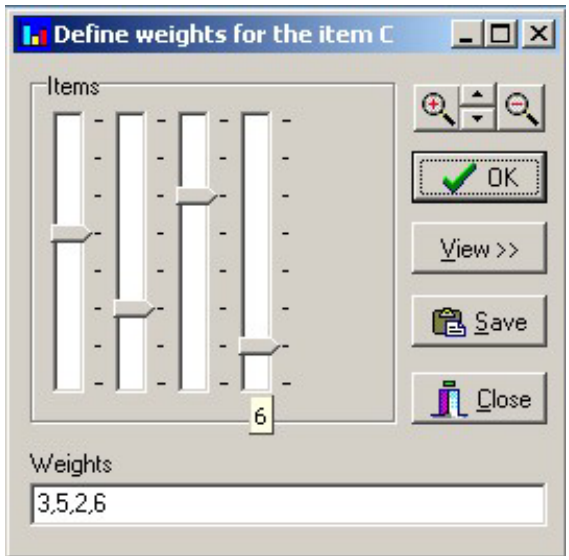


**Figure 4. Weights specification Window**

## V. DESIGNING A CR–TEST BENCH

Undoubtedly, the synthesis of a constrained-random generator is more sophisticated then it was presented in the

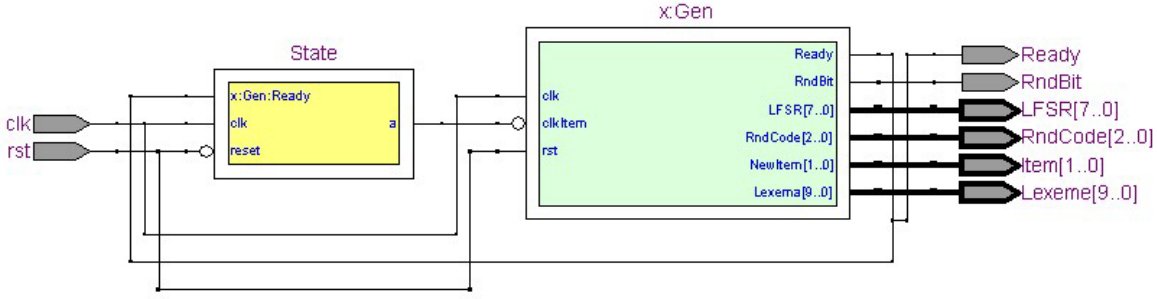


**Figure 5. Typical constrained-random generator RTL-diagram**

previous section. At first, it is needed to outline that the constrained-random generation is performed iteratively from the start item (as the item A in example) to the final item (as the item F) and so one: from A to A via F.

In VHDL–language this sentence is equivalent to the following loop-statement:

```
Gen: -- repeat until done
while TestActive loop
  case State is
    when A => · · · Next state <= B;
     . . .
    when K => · · · Next state <= A;
  end case;
end loop;
```

Secondly, the process of instruction "assembling" should be performed as simple as possible (more acceptable, in the same manner, i.e. containing same steps or operations). This specific peculiarity in the CR generation for synthesis is caused by the fact that a random value obviously can be "obtained" by any time of clock. One should mention that the probabilistic (weighted) transition is timing cost.

A simple scheme for instruction assembly is next proposed. Figure 6 shows the block-diagram of one stage conventional hierarchical "assembler".

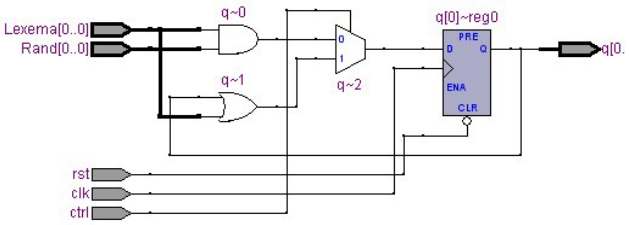


**Figure 6. Block diagram of DFF with multiplexing input**

Typically, at first, the masked value of LESR is assigned to flip-flop DFF, when signal CTRL is logic '0', and, secondly, if ctrl = '1' then lexeme's bit is "entered" in the DFF where mask value was equal to '0' (the designer must memorize this specific way of assembling the instruction!)

Thirdly, CR generator controls exclusively the generation of stimuli or in other words stimuli generation is controlled via UUV, e.g. when a verified unit generates an interrupt.

Lastly, the test-designer should implement a more complex generation process that is equivalent, for example, to a context-sensitive grammar where transitions are conditioned not only internally, but also by the external events (e.g. interrupt signals).

A CR test bench with the stochastic non-contextual generation may be implemented as following:

```
library ieee; use ieee.std_logic_1164.all;
use work.PIC16C5_pkg.all;
use work.RandGen_pkg.all;

entity TestBench is    port (…)
end entity;

architecture TestBench of TestBench is
begin
. . .
end;
```

Working folder contains the CR generator package PIC16C5_pkg, generated by proposed RandGen tools. Test-designers should only instantiate the generator and connect it to UUV. In the user test bench a UUV is instantiated by mapping the port interface, e.g. UUV: CR_PIC16 port map (rst=> rst, clk=> clk, Item=> Item, clkItem=> clkItem, Lexeme=> Lexeme, LFSR=> LFSR, RndBit=> RndBit, Ready=> Ready, RndCode=> RndCode).

## VI. TEST EXPERIMENTS AND RESULTS

After creating the input signal waveforms we proceed to simulations of the PIC16C5 test bench project. The aim of simulations is to estimate the toggle coverage that depends on test length.

Using Time End command of Quartus software we have scheduled the test bench simulation and recorded simulation coverage. Such test experiments were performed with different weight distributions. Figure 7 plots the obtained results. Curve 1 is plotted for the generator with equiprobable transition or uniform distribution on grammar rules; graphic 2 is plotted for the test experiment with a nonunoform weights, and graphic 3 is plotted for the test experiment with optimized weights. The optimization of distribution was performed applying the maximum entropy criteria as follows.

Let $I_n$ be the set of test sequences, e.g. instructions, test programs etc., generated at step $n$. From [7, 8] is known that a priori estimation of test length $n$ of a constrained-randomly generated stimuli depends on its probabilities $p_i$ as:

$$n \geq \frac{1}{\min_i \{p_i\}} \ln \frac{1}{1-\lambda}, \qquad (4)$$

where λ is a predefined confidence level; typically λ≥ 0,95.

From Equation (4) results that for same (unchanged) confidence level the test performance can be improved by maximizing the minimum value of $n$-step probabilities. This task can be solved in the following way. In (4) the distribution $P(I_n)=\{p_i \mid i \in I_n\}$ is calculated from Kolmogorov-Chapman matrix equation of Markov chain, obtained by adequately converting the corresponding stochastic grammar. Further, applying the dynamic programming (Bellman) and searching for maximization the entropy of finite Markov chain, the minimum probability at step $n$ can be found. In our analysis the MatLab software was used for calculation the $n$-step distribution and solving the optimization (i.e. maximin) problem. So, for the count of grammar loop, equal to 2 (depth of PIC16 stack) was found the optimal distribution over grammar rules, shown in (3).

## VII. CONCLUSION

In this paper were presented tools for generation of VHDL-templates and "ready" entities of the constrained-

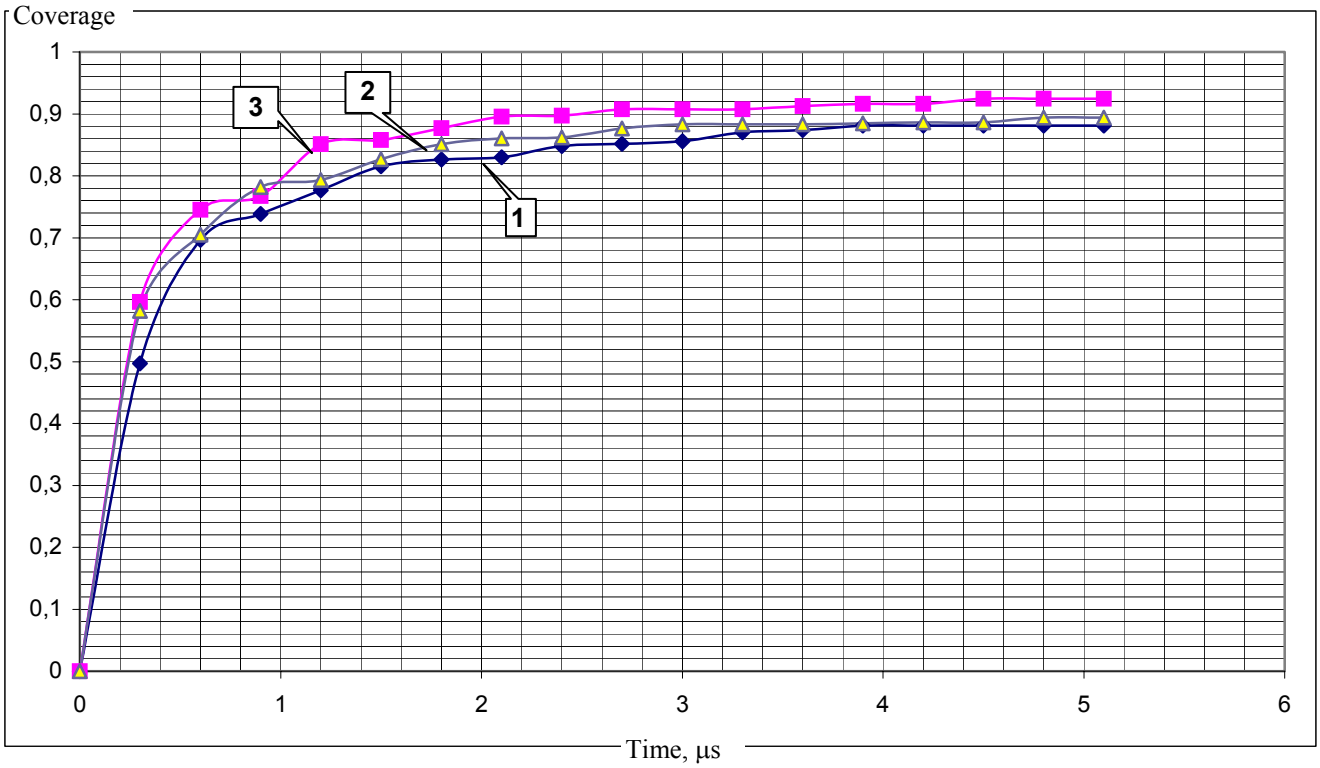


**Figure 7. Simulation coverage versus test length:**
**1 – uniform weights distribution;**
**2, 3 – nonuniform weights distribution.**

random generator for synthesis. The proposed tools can be viewed as an intermediate stage between built-in (V)HDL randomization constructs and resulting logic, implemented in PLD. In the paper was outlined that not only randomization constructs must be developed, but the technology of creating test benches that uses constrained-random generator must be also developed. In this sense, the formal grammar and stochastic process' fundamentals are helpful and well suited.

For the future work we intend to elaborate tools for synthesis of state machine diagrams associated with generation grammar.

Constrained-random test experiments with microprocessors and microcontrollers show a good correlation between the theoretical model and practical results. Thus, the test-designer is already equipped theoretically as well as practically for constrained random verification of complex logic design

REFERENCES

[1] J. Lewis, "Constrained Random Verification with VHDL", 2009 [Online available: www.SynthWorks.com].

[2] S.M.Thatte, and J.A.Abraham "Test generation for microprocessors", *IEEE Trans. Comput*. C-29, No. 6, 1980, pp. 429–441.

[3] L.-M.Wu, K.Wang, and C.-Y.Chiu, "A BNF-Based Automatic Test Program Generator for Compatible Microprocessor Verification", *ACM Trans. on Design Automation of Electronic Systems*, vol.9, No. 1, 2004., pp. 105-132.

[4] Gh.Bodean, "Microprocessor verification by syntactically controlled generation of the test programs", *Meridian ingineresc*, TUM, Kishinau, No.2, 2008, pp. 18-25.

[5] K.S. Fu, *Syntactic Methods in Pattern Recognition*, New York, Academic Press, 1974.

[6] J. Lewis, "Accellera VHDL-TC Extensions-SC Randomization," 2007. Available: http:// www.accelera.org/ apps/group_public/ download.php/ 905/Randomization-V1.pdf.

[7] I. F. Klistorin, V. I. Borshchevich, S. N. Filimonov, E. V. Morshchinin and V. F. Gushan, "Metrological support of information measuring systems for stochastic functional checkout systems of microprocessor devices*", Measurement Techniques*, Springer, New York, Vol. 32, No. 11, 1989, pp- 1048-1052.

[8] R. David**, *Random testing of digital circuits: Theory and applications*,** CRC Press, 1998.