2018 21st Euromicro Conference on Digital System Design

Program Generation Through a Probabilistic Constrained Grammar

Ondrej Cekan, Jakub Podivinsky, Zdenek Kotasek
Faculty of Information Technology, Brno University of Technology, Centre of Excellence IT4Innovations
Bozetechnova 2, 612 66 Brno, Czech Republic
Tel.: +420 54114-{1361, 1361, 1223}
Email: {icekan, ipodivinsky, kotasek}@fit.vutbr.cz

***Abstract*—The paper introduces a probabilistic constrained grammar which is a newly formed grammar system for use in the area of test stimuli generation. The grammar extends the existing probabilistic context-free grammar and establishes constraints for grammar limitations. Stimuli obtained through the proposed principle are used in the functional verification of a RISC processor and coverage metrics are evaluated. The detailed information about the construction of an assembly code for processors is described, as well as the experimantal results with the implemented generator. Experiments show the expressive power of the probabilistic constrained grammar and achieved code coverage in the verification of the processor. The grammar system demonstrates that is very suitable for an assembly code generation and universal use in the area of test stimuli.**

Keywords*—Probabilistic Constrained Grammar, Probabilistic Context-Free Grammar, Stimulus, Constraint, Functional Verification

I. Introduction

Nowadays, electronic circuits become more and more complex due to new technologies of production and many different components are merged into one chip. It causes a problem with testing and verification of the whole system too. The classical approaches of testability fail, as well as formal techniques for the verification of the large systems. The emphasis on the quality of a proposed system is also still rising, therefore, thorough verification of the system has to be done. Due to this fact, the functional verification technique was designed to accelerate the verification of the correct behaviour of complex systems [1].

In the functional verification, the system inputs are set while its outputs are monitored. The functional verification is based on two systems which are tested in parallel with the same input data (stimulus). The first system is the hardware implementation of a device typically described in HDL (Hardware Description Language) [2], known as DUT (Device Under Test) which verifies its correctness due to the given specification. The second system is a model of the verified system which corresponds to the same specifications and is typically implemented in a different programming language. The model is known as the golden model. The same stimulus is brought to the input of these two systems which is typically obtained from a stimulus generator. Both systems are simulated. The output of the functional verification is the result of comparing the outputs of both systems on equality and also the information about the coverage of the key system functions [3]. In the context of the simulation environment, the metrics and conditions (key functions) which should be monitored can be defined. Therefore, coverage is an important metric in this process. It defines the percentage value which represents the level of the system verification, and how well input stimuli cover the behaviour of the system.

In our research, we focus on the stimuli generation which can be used in the functional verification. We benefit from the grammar systems which allow us to formally define and generate any language through the application of production rules. We extend the grammar system through special constraints which restrict the application of the production rules in the grammar. In this paper, we show stimuli generation of a valid assembly code for a RISC processor [4] and we measure the coverage value in our experiments through the functional verification.

The text of the paper is structured as follows. Section 2 describes our previous research which this work is based on. The state of the art in the area of stimuli generation is described in section 3. In section 4, a probabilistic constrained grammar with the design of input structures is described. Section 5 describes the process of encoding instructions into the probabilistic constrained grammar. In section 6, a method of generating stimuli through our architecture is demonstrated. The experiments with the generation of assembly programs through the proposed principle and conventional approaches are presented in section 7. Finally, in section 8, we summarize the results in the conclusion.

II. Previous Research

In our previous research, we concentrated on:

- A universal architecture of stimuli generation, which is based on two input structures, was developed. [5] The first structure called *Format* contains the information about the format of the stimulus. The second structure called *Constraints* defines the restrictive conditions for stimulus format and prescribes how the stimulus should be created, because only a subset of all possible solutions is valid for the given system. Based on these structures, valid stimuli are generated. We utilize this architecture in this work where a stimulus format is described through a probabilistic context-free grammar and together with constraints, it defines our new grammar system - probabilistic constrained grammar. The extended architecture for the use in grammar systems can be seen in Fig. 1.

978-1-5386-7377-5/18/$31.00 ©2018 IEEE
DOI 10.1109/DSD.2018.00049

CPS Conference Publishing Services

- Specific input structures (no general) for describing assembly programs for RISC and VLIW processors, and input structures for generating random mazes for the robot controller were designed [6].
- A stimuli generator based on the described architecture which generates a defined set of stimuli on the basis of the problem of constraint solving [7] was implemented.

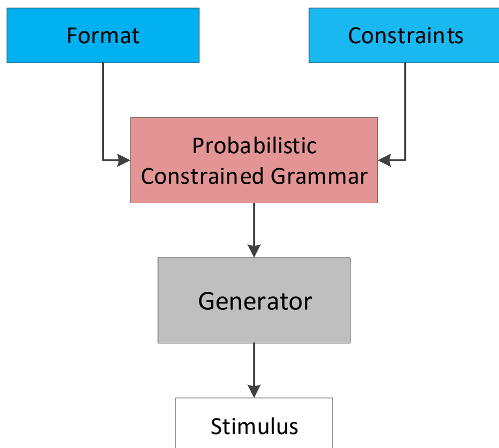

Fig. 1: The versatile architecture of stimuli generation.

III. Related Work

An assembler code is mainly used as a randomly generated program for processors, because it does not require a compiler, and therefore, offers a full control over the operations and registers of a processor. These programs are typically described by several input blocks that are designed for a specific processor, and therefore, they can not be used for another type of processor or different system. As an input block, a description of the processor instruction set (ISA) [8] is used and is combined with either a VHDL processor description [9] or some micro-architecture elements [10]. Another approach is through a specially designed library [11] that defines all possible combinations of instructions and operations that are valid for the given processor and the resulting program is obtained using the genetic algorithm [12]. The generation approach using the abstract processor model has been also described in the literature [13].

These approaches are quite complex, the definition of stimuli is complicated and dependent on a specific processor. The use of such generator of stimuli is very time-consuming, because the input block definition is quite extensive and based on proprietary formats that combine detailed information about the processor architecture and the semantics of each instruction. The generators are also limited to the specific processor and can not be used for any other.

As a universal stimuli generator, we can mention MicroGP tool [14] which does not only generate stimuli but it also finds the most optimal solution of hard problems. The architecture of this tool is composed of 3 separated blocks: an *evolutionary core*, a *problem definition* (an instruction library) and an *external evaluator*. The evolutionary core and external evaluator are the blocks for the optimization process. The optimization process of the MicroGP tool is not in this paper discussed and is only utilized the block of *problem definition*. In the commercial sector, there are different program generators (e.g. GenesysPro from IBM company [15]), however, they can not be obtained and compared with them.

In this paper, we focus on universal stimuli generation which is also suitable for assembly code generation. Through the probabilistic constrained grammar, we are able to define the desired instructions of the processor in a consistent way. This work represents a generalization of our previous research. Our approach of stimuli generation is compared with commercial program generator from the Codasip company [16] and with the MicroGP tool and the coverage of the process of the functional verification is compared.

IV. A Probabilistic Constrained Grammar

Grammar is an instrument for the representation of any language [17]. It is a generative system that can represent the finite and infinite languages. The grammar uses two disjoint finite alphabets: 1) the set N of non-terminal symbols, and 2) the set T of terminal symbols. The non-terminals serve as the auxiliary variables which are substituted for the non-terminal or terminal strings through production rules and the substitution takes place until the string contains only terminal symbols. Then, the sentence of the defined language is generated.

We introduce the definition of a new grammar system which combines existing context-free grammar with a constraint satisfaction problem (CSP) [18]. CSP deals with the assignment of values from a particular domain with respect to restrictive conditions. We have described the new grammar system as a Probabilistic Constrained Grammar which is pair G:

`G = (H,C); where:`

H	is a probabilistic context-free grammar.
C	is a ordered list of constraints for the grammar H.

A probabilistic (stochastic) context-free grammar [19] is a basic context-free grammar into which probabilities for the application of production rules were delivered. It is the 5-tuple:

`H = (N,T,R,S,P); where:`

N	is a finite set of non-terminal symbols.
T	is a finite set of terminal symbols, $N \cap T = 0$.
R	is a finite set of production rules with form $A \rightarrow \alpha$, where $A \in N$ a $\alpha \in (N \cup T)^*$.
S	is the starting non-terminal.
P	is a finite set of probabilities for production rules.

The constraints represent the definition of the CSP and restrict the grammar in the application of the production rules. The non-terminal symbols N of grammar H can represent

variables that are constrained through probabilities P in the application of the rules. The set of production rules R, where the non-terminal X is on the left side of a rule, represents the domain of values for the given non-terminal. The constraint is the 5-tuple:

C = (R_S,R_D,P,[R_E],[O]); where:

- R_S is the identifier of the rule which calls this constraint.
- R_D is the identifier of the rule for which the probability is changed.
- P is the new probability value.
- R_E (optional) is the identifier of the rule, the application of which causes the abolition of the constraint.
- O (optional) is the count of derivations of R_E rule before abolishing the constraint.

The constraints limit the application of production rules for a given non-terminal and, therefore, restrict all possible strings in a given formal language. The probabilistic context-free grammar itself is a statical definition of a language, while the addition of the constraints will cause a dynamic change of the generated language during the application of production rules. It is a certain analogy of programmable grammar; however, it depends on the context that has been generated so far.

The implementation of the generator based on the new grammar performs the application of the rules from the starting non-terminal with leftmost derivations. After the application of any rule, the constraints for the relevant rule are triggered and new probabilities are set for the given rules.

V. Encoding Instructions into the Grammar

In the context of coding instructions of the processor into a probabilistic context-free grammar, we introduce three conditions which have to be ensured against the standard definition of the grammar.

Firstly, the production rules may not be strictly defined with probability values in which they can be applied. In the case that the probability for a rule is missing, it will be calculated as $100\% - \sum definedProbabilities$ for the given non-terminal. In the case where there is no definition of probability for multiple rules, the probability for each rule will be the same and will be calculated as $(100\% - \sum definedProbabilities)/numberOfRulesWithoutProbability$ for the given non-terminal. Through this, the explicitness for the application of the rules without a strictly defined probability is defined. The probability in most cases will not be defined, because we have the goal of gaining the same probability for almost every rule because of the large number of combinations for generating an instruction in the program.

Secondly, the probabilities will not be calculated from the training data set (as is in the typical application of the grammar), but they will be determined by an engineer on the basis of their knowledge about the processor. The main task is to limit a certain group of instructions in order to generate them in the minimum. This group can be represented by jump instructions. Their excessive generating will cause low utilization of the program code. The utilization of the program code is defined by instructions which are physically executed on the processor. Through the grammar definition we do not describe semantics of the instructions (it is not the aim of this principle), but only syntaxes. Therefore, we are able to generate only forward jumps (i.e. their execution is independent of the previous instruction sequences).

Thirdly, production rules which have some constraints must be clearly identifiable (i.e. they must have an identifier). Assigned probabilities of certain rules of the grammar will change during the stimulus generation. It is needed to identify the rules in which the probability will be dynamically changed based on the previously used rules. The rules are typically labelled numerically, but we will use a combination of alphanumeric characters.

A. The Process of Encoding Instructions

Processor instructions should be divided into several groups depending on the type of the instruction. Each group is defined by a custom non-terminal into which it is possible to get from the starting symbol. Each group has a defined probability which is increased/decreased on the basis of the type and the count of instructions. The arithmetic instructions will typically have a higher probability than jump instructions. Based on the format of the instruction, each group is subdivided into another non-terminal which brings together the same format of instructions. For example, the arithmetic instructions which work with two register operands will be in a different group than the arithmetic instructions which work with a register and immediate operand. In the next step, each instruction is defined by using a template that is composed of non-terminal and terminal symbols. Non-terminal symbols are already replaced to specific registers and operations which create the actual instruction of the program.

B. The example of encoding instructions

This example shows the part of the assembly probabilistic context-free grammar for the Codix Cobalt processor [16] developed in the Codasip company [16]. This is a 32bit RISC processor which contains around 60 types of instructions. Consider *S* as the starting grammar non-terminal. The instruction set of the processor can be split into 5 groups - arithmetic (ARITHM), memory (MEMORY), conditional (CONDIT), jump (JUMPS) and other (OTHERS) instructions. The definition of these production rules, including implied probabilities, is as follows:

```
S → ARITHM(50%)|MEMORY(20%)|CONDIT(15%)|
    JUMPS(5%)|OTHERS(10%)
```

The set probabilities are very important to achieve the highest coverage in our experiments in the fastest possible time. These probabilities were experimentally found as the best setting. Other settings will cause slower coverage convergence.

In the case that at the end of these rules we define again the starting non-terminal, we get a cyclic instruction generation. We chose a group of arithmetic instructions that can be divided into 6 subgroups with different formats which

specify other non-terminals. This includes a subset of instructions using two registers as operands (ARITHM_RR), register and immediate operand (ARITHM_RI), three registers as operands (ARITHM_THREE), instructions for sign extension (ARITHM_EXT), assignment instruction (ARITHM_ASS), and instructions for assigning numbers into the upper half of the immediate operand (ARITHM_LUI). The rules defining these groups production the non-terminal to a specific syntax of the given instruction according to its definition. Register and number values are still hidden behind other non-terminals. Several examples of the definitions of these rules are as follows:

```
ARITHM → ARITHM_RR|ARITHM_RI|
          ARITHM_THREE|ARITHM_EXT|
          ARITHM_ASS|ARITH_LUI
ARITHM_RR → DST = ARITHM_NAME SRC, SRC
ARITHM_RI → DST = ARITHM_NAME SRC, IMM
ARITHM_NAME → add|sub|add|or|xor|shl|shr
DST|SRC → r0|r1|r2|r3|r4|r5|r6|...|r31
...
```

At first sight, it seems that *DST* and *SRC* non-terminals can be represented by the same non-terminal because they specify the same register values and, therefore, we may not have two different definitions of the rules. In fact, it requires the differentiation between operands of the instruction because constraints for the rules deriving the *DST* non-terminal will be different than constraints for *SRC* non-terminal - for example, in order to preserve the latency between instructions. The entire selected branch of the derivation tree for the arithmetic operation *ADD* is shown in Fig. 2. It is obvious that through simple adjustments we are also able to generate a binary representation of the assembly program.

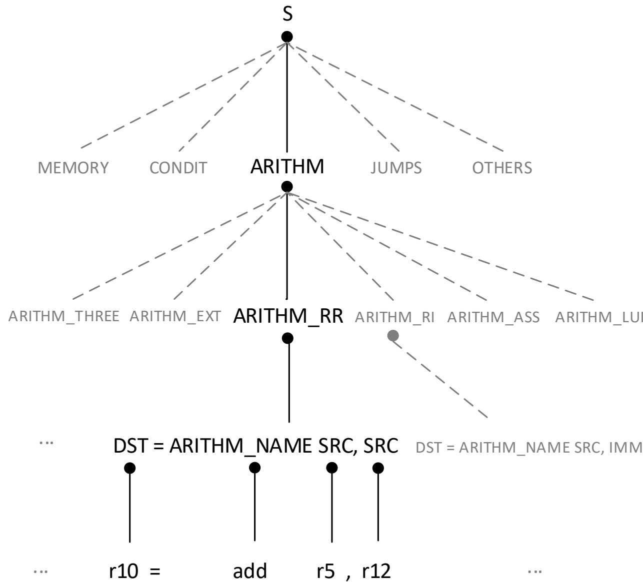

Fig. 2: The derivation tree for ADD instruction which is composed of destination register, instruction name, and two source registers.

C. The example of constraints for instructions

For the clarity, we use the keyword *C* before the definition of constraints.

1) Latency ensuring: Let us consider a simplified probabilistic context-free grammar *H* with rules, where EOL marks a new line:

```
S → DST = add SRC SRC EOL S
dr1: DST → r1
dr2: DST → r2
dr3: DST → r3
sr1: SRC → r1
sr2: SRC → r2
sr3: SRC → r3
eol: EOL → \n
```

Through the constraints we want to achieve that the *add* instructions will have latency on two. It means that the generator cannot use the result *DST* as the source operand *SRC* in one following instruction; therefore, the result is not stored in the register yet. The constraints will be defined in the following way:

```
C(dr1, sr1, 0, eol, 2)
C(dr2, sr2, 0, eol, 2)
C(dr3, sr3, 0, eol, 2)
```

It is able to read the first constraint in the following way: after application of the *dr1* rule, the probability for the *sr1* rule is set to 0, and after two applications of the *eol* rule, the probability for the *sr1* rule is set back to the default value (33.33%).

2) Absolute forward jumps: Let us consider a simplified probabilistic context-free grammar *H* with rules:

```
S → ADD S(80%)|JUMP S(20%)
T → ADD T|ADD
ADD → r3 = add r2 r1 EOL
jmp: JUMP → jump NAME EOL T NAME:
EOL → \n
nj1: NAME &→ STR1
nj2: NAME &→ STR2
nj3: NAME &→ STR3
```

Then the jump instruction *jump NAME* will be generated by using the *jmp* rule, including its label marked by *NAME*: non-terminal. Between the jump instruction and its label any other instruction can be placed, except jumps. Both *NAME* non-terminals must be derived into the same string; therefore, the leftmost derivation as a classic variant cannot be used. For this case, a special derivation characterized with &→ mark is used. This ensures that all *NAME* non-terminals are derived through the same selected rule. The use of the following constraints ensures the reduction of the probability to zero after the application of a randomly selected label and thus the uniqueness of the label in the whole program.

```
C(nj1, nj1, 0)
C(nj2, nj2, 0)
C(nj3, nj3, 0)
```

3) Number of program instructions: It is possible to set the number of instructions in a program through the proper settings of the production rules and constraints. Consider a probabilistic constrained grammar *G* with rules and constraints:

```
start: S → START
end: START → END(100%)
START → ADD START
ADD → r3 = add r2 r1 EOL
END → nop
eol: EOL → \n

C(start,end,0,eol,1000);
```

The key settings of the number of instructions lies in the addition of a simple production rule that ensures the application of the one defined constraint. It can be seen that current configuration of the production rules will write only one *nop* instruction in the program by application of the *end* rule. The invocation of the constraint disables the application of the *end* rule for the 1,000 productions and thus 1,000 of *ADD* instructions will be generated before the adjusted probability is removed. After that, the one *nop* instruction will also be generated.

VI. Random Program Generation

The detailed architecture of the stimuli generation based on the grammar system is shown in Fig. 3. In the figure, there are two input structures which contain the probabilistic context-free grammar (Format) and Constraints for restricting the grammar rules. In the definition of structures, we use the Jinja2 templating system [20], [21] which allows us to define the cycles, branches and other special macros that simplify the entry of production rules of the grammar. These structures are preprocessed in the first step. Preprocess performs the expansion of the Jinja2 macros and the result of this process are the Extended structures of the probabilistic context-free grammar and constraints which already contain the complete definitions of the production rules and constraints necessary to ensure the completeness and validity of the generated program. The extended structures form the final Probabilistic constrained grammar which is processed by the Generator core. It performs the application of the rules from the starting non-terminal with leftmost derivations. After the derivation of any rule, specific constraints are triggered and new probabilities are set for selected rules. The demonstration of the *NAME* non-terminal definition for the set of names for label of jump instructions through the library Jinja2 follows:

```
{% for i in range(1,100) %}
    NAME &→ STR{{i}}
{% endfor %}
```

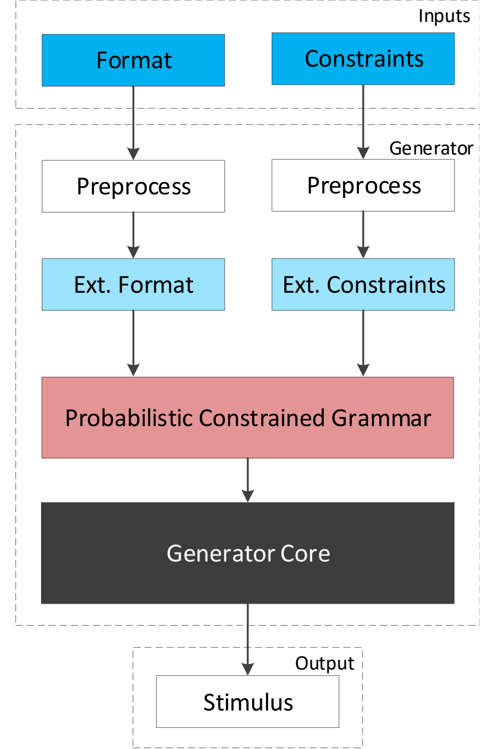

Fig. 3: The detailed architecture of the stimuli generation based on the grammar system.

A. Expressive power of the new grammar

The probabilistic constrained grammar alters the behaviour of the original context-free grammar, and offers completely new fields in the application area. It must also be mentioned that the expressive power of the newly formed grammar is displaced towards context languages which is a necessary condition to generate a valid test stimulus. In the case of the program generation, the context is necessary in order to preserve the correct order of some assembly instructions, uniqueness of jump instructions labels, etc. The proof that the expressive power is changed from the context-free language to the context language, or at least to the partially context language, shows the following language:

$$L(G) = \{ a^n b^n c^n : n \geq 1 \}$$

Strings which belong to the given language *L(G)* have a non-zero length and are always sequences of *a* characters followed by equally long sequences of *b* characters and *c* characters. This language is context sensitive and thus cannot be generated through context-free grammar due to its inability to ensure the application of the same number of production rules. The context-free grammar which can be defined for the similar language can look like:

```
S → ABC
A → aA|a
B → bB|b
C → cC|c
```

VII. Experimental Results

In our experiments, we focused on two criteria. The first criterion has its origin in the language theory and its aim is to determine the expressive power of the new grammar. The second criterion is the practical use of generated test stimuli in the field of the functional verification.

TABLE I: The total code coverage in the functional verification for the generators for 100 programs.

Number of instructions	100	200	500	1000	2000	5000	10000	15000	20000	25000
USG	61.01%	72.33%	76.29%	79.35%	82.13%	84.50%	85.81%	86.45%	86.91%	87.26%
commercial	60.90%	72.47%	76.35%	79.48%	81.88%	84.04%	85.16%	85.63%	86.23%	86.56%
microGP	60.92%	71.88%	75.54%	78.59%	81.02%	82.92%	83.87%	84.37%	84.69%	84.91%
RISCG	60.94%	72.40%	76.13%	79.34%	81.76%	84.21%	85.44%	85.98%	86.02%	86.31%

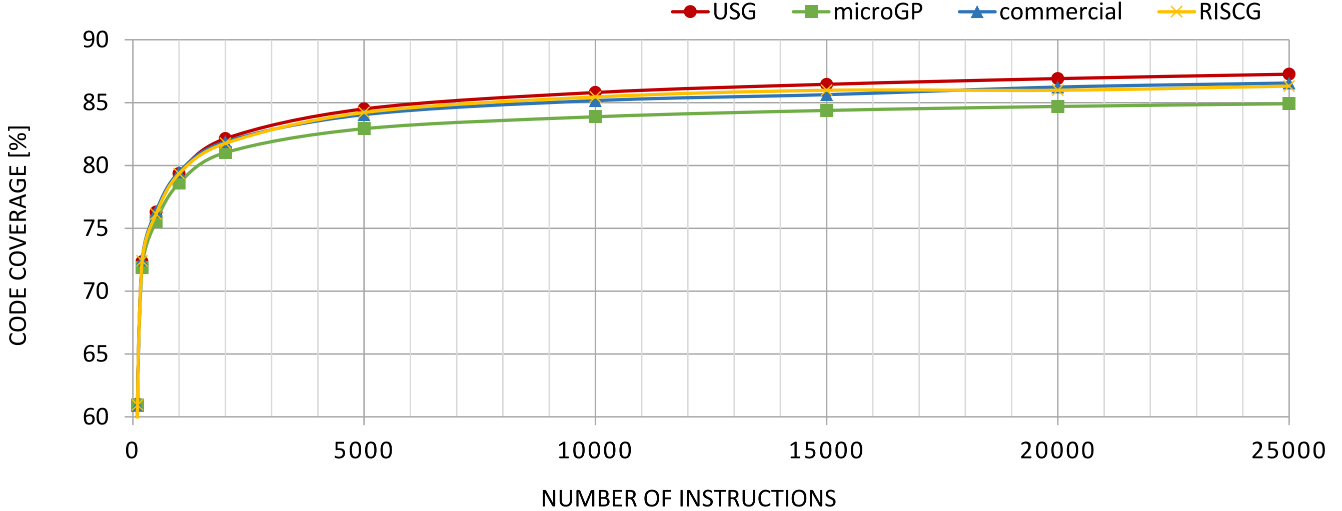

Fig. 4: The comparison of achieved code coverages in the functional verification for 100 programs.

with language:

$$L(G) = \{ a^m b^n c^o : m, n, o \geq 1 \}$$

The context-free grammar can be modified for generating the language $a^m b^n c^n$ where m, n $\geq$ 1, but it is still not the previously defined language. For the previously defined language, the context sensitivity through the probabilistic constrained grammar is needed. In the following example, we show the minor modification of the *L(G)* grammar which is written through the probabilistic context-free grammar:

```
s: S → ABC
a: A → aA
ae: A → ϵ(100%)
b: B → bB
be: B → ϵ(100%)
c: C → cC
ce: C → ϵ(100%)
```

The number of applications of A, B, and C non-terminals is achieved by adding three constraints which ensure this functionality and establish context sensitivity. The *rn* number of applications of production rules can be randomly generated through a templating library [20] and the number value is forwarded to the last parameter of each constraint.

```
{% set rn = random(1000)+1 %}
// rn can be set to any value
C(s,ae,0,a,{{rn}});
C(s,be,0,b,{{rn}});
C(s,ce,0,c,{{rn}});
```

B. *Coverage in the functional verification*

Grammar systems provide a new dimension in input test stimuli generation. For this reason, we decided to perform an experiment by achieving the highest coverage in the functional verification on the Codix Cobalt processor. In our experiments, we focus on the *code coverage*. It measures the system source code through typical metrics like statements, branches, expressions, conditions, and states. The main task of the experiment is to verify the influence of stimuli generated using the grammar system (marked as USG) on their quality and achieved coverage in comparison with the microGP tool, the commercial generator from Codasip and our previously developed program generator for RISC processors (marked as RISCG [6]). It should be mentioned that no optimization process has been activated for the tools (e.g. genetic algorithm) to compare the results with each other.

In the experiment, we generated 100 programs with a different number of assembly instructions through the mentioned generators. All programs have been verified by the functional

verification on their validity and the total code coverage has been measured. The result of the experiment can be seen in Table I and Fig. 4.

From the results, it can be seen that for a small number of instructions in the program, the coverage of all generators is almost identical. This is accomplished by the fact that a small number of different instructions trigger a series of transactions that are executed, and therefore, the code coverage is suddenly raised to 70-80%. For the higher number of instructions in the program, the instruction sequence is very important which invokes transactions occurring only in a certain state of the processor and sequence of instructions. For this reason, the coverage is rising relatively slowly and for 25,000 instructions in 100 programs, the maximum possible code coverage is approximately achieved.

The maximal total code coverage was 87.26% for our USG generator for 100 programs with 25,000 instructions (25,000 instructions is the maximum for the simulation tools). Next in row was the commercial generator, our previously developed generator RISCG and MicroGP tool. It can be stated that the test stimuli generation using the grammar system is an appropriate and effective way of stimuli generation. The generation time spent to create the stimulus was 1.1 second in case of USG, 0.7 second in case of RISCG, 1.5 second in case of commercial tool and 43 seconds in case of MicroGP tool for 25,000 instructions.

VIII. Conclusions and Future Research

In this paper, our research in the field of random test stimuli generation was presented. For the definition of the stimuli, we used two input structures which defined the format of the stimuli and constraints. We introduced the new grammar system - a probabilistic constrained grammar which was practically implemented and verified in assembly code generation. The valid stimulus was obtained through constraint definitions which restrict the defined grammar in the application of production rules. The expressive power of the probabilistic constrained grammar was proven to be higher than the expressive power of context-free languages. The experiment showed that the test stimuli generation using the grammar system is competitive and more profitable than other way.

The presented approach can be used for stimuli generation of various systems. The approach was also applied on the robot controller implemented into FPGA. The stimuli were represented by mazes in which the robot controller searched the path from the start to goal position (more information can be found in [22]). In our future research, we are working on an on-line solution of the generator and we will direct our efforts towards creating a methodology for using the proposed approach in the area of stimuli generation.

Acknowledgement

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science - LQ1602 and BUT project FIT-S-17-3994.

References

[1] A. Meyer, *Principles of Functional Verification*. Amsterdam: Elsevier Science, 2003.

[2] P. Ashenden. (1990 [cit. 2015-01-02]) The vhdl cookbook [online]. Adelaide. [Online]. Available: http://www.ics.uci.edu/ alexv/154/VHDL-Cookbook.pdf

[3] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *Design and Test of Computers, IEEE*, vol. 18, no. 4, pp. 36–45, May 2001.

[4] "Risc principles," in *Guide to RISC Processors*. New York: Springer, 2005, pp. 39–44.

[5] O. Cekan, M. Simkova, and Z. Kotasek, "Universal pseudo-random generation of assembler codes for processors," in *Proceedings of The 4th Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale*. COST, European Cooperation in Science and Technology, 2015, pp. 70–73. [Online]. Available: http://www.median-project.eu/wp-content/uploads/18_IV-2_median2015.pdf

[6] J. Podivinsky, O. Cekan, M. Simkova, and Z. Kotasek, "The evaluation platform for testing fault-tolerance methodologies in electro-mechanical applications," *Microprocessors and Microsystems*, vol. 39, no. 8, pp. 1215 – 1230, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933115000630

[7] R. H. C. Yap, "Constraint processing by rina dechter, morgan kaufmann publishers, 2003, hard cover: Isbn 1-55860-890-7, xx + 481 pages," *Theory Pract. Log. Program.*, vol. 4, no. 5-6, pp. 755–757, Sep. 2004. [Online]. Available: http://dx.doi.org/10.1017/S1471068404222189

[8] D. A. Patterson, "Reduced instruction set computers," *Commun. ACM*, vol. 28, no. 1, pp. 8–21, January 1985.

[9] J. Hudec, "An efficient technique for processor automatic functional test generation based on evolutionary strategies," in *Proceedings of the ITI 2011, 33rd International Conference on Information Technology Interfaces*, May 2011, pp. 527–532.

[10] V. Belkin and S. Sharshunov, "Isa based functional test generation with application to self-test of risc processors," in *Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE*, April 2006, pp. 73–74.

[11] F. Corno, E. Sanchez, M. Reorda, and G. Squillero, "Automatic test program generation: a case study," *IEEE Design and Test of Computers*, vol. 21, no. 2, pp. 102–109, March 2004.

[12] S. N. Sivanandam and S. N. Deepa, *Introduction to Genetic Algorithms*, 1st ed. Springer Publishing Company, Incorporated, 2007.

[13] F. Corno, M. Reorda, G. Squillero, and M. Violante, "A genetic algorithm-based system for generating test programs for microprocessor ip cores," in *Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2000)*. IEEE Computer Society, November 2000, pp. 195–198.

[14] G. Squillero, "Microgp—an evolutionary assembly program generator," *Genetic Programming and Evolvable Machines*, vol. 6, no. 3, pp. 247–263, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10710-005-2985-x

[15] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-pro: innovations in test program generation for functional processor verification," *IEEE Design Test of Computers*, vol. 21, no. 2, pp. 84–93, Mar 2004.

[16] Codasip. (2016) Codasip — enabling the internet of extraordinary things. [Online]. Available: http://www.codasip.com

[17] A. Meduna, *Formal Languages and Computation: Models and Their Applications*, 1st ed. Boston, MA, USA: Auerbach Publications, 2014.

[18] V. Kumar, "Algorithms for constraint satisfaction problems: A survey," *AI MAGAZINE*, vol. 13, no. 1, pp. 32–44, 1992.

[19] R. Giegerich, *Introduction to Stochastic Context Free Grammars*, J. Gorodkin and L. W. Ruzzo, Eds. Totowa, NJ: Humana Press, 2014.

[20] A. Ronacher. (2014) Jinja2 (the python template engine). [Online]. Available: http://jinja.pocoo.org/

[21] M. Lutz, *Learning Python*, 2nd ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2003.

[22] J. Podivinsky, O. Cekan, J. Lojda, M. Zachariasova, M. Krcma, and Z. Kotasek, "Functional verification based platform for evaluating fault tolerance properties," *Microprocessors and Microsystems*, vol. 52, pp. 145 – 159, 2017.