2017 Euromicro Conference on Digital System Design

A Probabilistic Context-Free Grammar Based Random Test Program Generation

Ondrej Cekan, 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, 1223}
Email: {icekan, kotasek}@fit.vutbr.cz

***Abstract*—The aim of this paper is to show the use of a probabilistic context-free grammar in the domain of stimulus generation, especially random test program generation for processors. Nowadays, the randomly constructed test stimuli are largely applied in functional verification to verify the proper design and final implementation of systems. Context-free grammar cannot be used by itself in this case, because conditions for instructions of the program are changing during the generation. Therefore, there is a need to introduce additional logic in the form of constraints. Constraints guarantee the continuous changes of probabilities in the grammar and their application in order to preserve the validity of the program. The use of the grammar system provides a formal description of the stimuli, while the connection with constraints allows for the wide use in various systems. Experiments demonstrate that this approach is competitive with a conventional approach.**

Keywords*—*Probabilistic Context-Free Grammar, Random Test Program Generation, Stimulus, Constraint

I. Introduction

Electronic circuits are presently used in many facilities, therefore, people regularly meet them in their lives. Reliability in terms of hardware components, and also in terms of the software design and solution, plays an important role in these systems. The incorrect behaviour which may occur in a system during the operation could be very costly for manufacturers and human lives can be endangered, especially in critical applications. For these reasons, systems under development must be tested thoroughly for the purpose of eliminating design and implementation errors during development. The usual and unusual combinations of input values that can occur in the system must be taken into account. The complexity of the system is continually growing, as well as the complexity associated with thorough verification of the system functions which are increasing [1]. It is not difficult to test simple systems manually. For more complex systems, manual testing is very time consuming. In addition, previously developed formal techniques for the verification of large systems have failed. Therefore, the technique called functional verification was developed.

Functional verification [2] is the activity of checking the correctness of the system according to its specification. In this activity, two systems are tested in parallel with the same input data (stimulus). At present, the stimulus is obtained from a generator and is constructed randomly.

In our research, we benefit from the grammar systems which allow us to formally define and generate any language. This language forms desired stimuli for the given system. In this paper, we show that is possible to generate an assembly code for a processor through the probability context-free grammar with our extension. An innovation that we bring to this grammar is the dynamic change of probabilities during the generation of the language through a special constraint definition.

The text of the paper is structured as follows. Section 2 describes the state of the art in the area. In section 3, the aim of our research is presented. A probabilistic context-free grammar with the process of instructions encoding is described in section 4. Section 5 describes the purpose of constraints. In section 6, a method of generating stimuli is demonstrated. The experiments with the generation of assembly programs through the proposed principle and conventional approach is presented in section 7. Finally, in section 8, we summarize the results.

II. Related Work

The current research in the field of program generation deals with the automatic generation of an assembly code for a specific processor. The programs are obtained from several input blocks that describe the processor. These input blocks are typically designed for the given type of processor and, therefore, it loses the flexibility of the solution for wider use. The description of the instruction set (ISA) [3] of the processor is used as an input which is combined with another description. The paper [4] uses certain elements of the processor micro-architecture as the second description. The paper [5] uses the VHDL description (VHSIC Hardware Description Language) of the processor as the second description. Another work [6] that automatically generates programs for processors, utilizes self-designed instruction library which describes the assembler syntax of each instruction and valid operand combinations. Together with a genetic algorithm (GA) [7], the resulting program is constructed. The work [8] shows the generation of assembly programs on the basis of an abstract model of the processor. According to this abstract model, programs are formed by means GA.

A significant disadvantage in the above mentioned solutions is seen due to the complexity of the stimuli description and the inability of using the generator in various systems other than the selected processors. The presented solutions are based on proprietary formats that work with detailed information about a selected processor and it is very time consuming to use such generators.

978-1-5386-2146-2/17 $31.00 © 2017 IEEE
DOI 10.1109/DSD.2017.26

From among versatile solutions which deal with a random test stimuli generation, the MicroGP tool can be mentioned [9]. Originally it was an assembly program generator for testing microprocessors, but later it was used for a wider range of problems. MicroGP uses GA for finding the 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 generates a population of individuals and performs the optimization process. The problem definition contains macros of instructions for valid assembly code generation in the case of the processor. The external evaluator simulates the program and provides the feedback to the evolutionary core.

In this paper, we present the solution which uses context-free grammars that allow us to define stimuli for the selected system in a consistent way. This work represents a generalization of knowledge gained from our previous research [10] where we generated assembly programs based on two proprietary input structures which defined the format of the stimulus and its restrictive conditions. The comparison of our principle will be done with MicroGP which is similar in versatile functionality but different in the used approach.

III. The Goals of the Research

We have two main goals in our research:

1) *To develop a stimuli generation framework for various systems.*
2) *To develop a methodology for using this framework in stimuli generation.*

Under the concept of stimulus generation we understand generating randomly constructed input test data that determine the behaviour of the system. In the case of a processor, the input stimulus is a program which determines its computing operation. In the case of a robot controller, input stimulus is a maze that the robot goes through. This random stimulus creates new circumstances which the system must solve.

The first goal of our research is described in this paper. It represents a generalization of the gained knowledge and definition of a universal description of stimuli which is based on the grammar system. The description of stimulus through probabilistic context-free grammar with constraints provides a formal representation of the stimulus and a possibility of its use in various systems. In our architecture, we use the previously designed schematic of the universal generation (see Fig. 1).

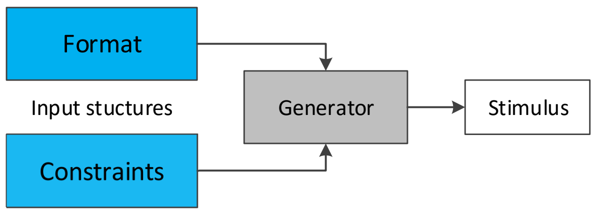

Fig. 1: The architecture of the universal stimuli generation.

The second main goal represents our long-term direction which we intend to achieve in our research.

IV. A Probabilistic Context-Free Grammar

Probabilistic context-free grammars [11] were introduced in bioinformatics where they have been used for modelling RNA structures. Their possible usage was found in other areas, especially in the field of natural language processing or creating a programming language. These grammars are based on fundamental context-free grammars where into the rewrite rules the probabilities that a rule can be applied are delivered. Probabilistic context-free grammar is the quintuplet:

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

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

For the probability in the context-free grammar, the following Definition 1 must be applied.

Definition 1: Consider a probabilistic context-free grammar *G*. For each rule *r* in grammar *G*, the transition probability π_r is defined. For each non-terminal $A \in N$ with its rewrite rules $r_1{:}A \rightarrow \alpha_1$, $r_2{:}A \rightarrow \alpha_2$, ..., $r_k{:}A \rightarrow \alpha_r$ the following rule must be applied: $\sum_{i=1}^{k} \pi_{ri} = 1$.

A sample of writing probability for individual rules (the character | means 'or') is:

```
S → AS(90%)|A(10%)
A → aBc(70%)|abc(30%)
B → bb(100%)
```

A. *Encoding Instructions Into the Grammar*

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 rewritten to specific registers and operations which create the actual instruction of the program.

V. Constraint Definitions

Constraints represent restrictions and limitations for derivation of rewrite rules and their application will change defined probabilities for specific rules. These constraints are defined as a function call without a return value, so it is a command. The constraint is defined as the quintuplet:

`cons(`R_S`,`R_D`,P,[`R_E`],[C]); 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.
C	(optional) is the count of derivations of R_E rule before abolishing the constraint.

The task of the constraint is to set the probabilities during the generation process so that the result is a valid stimulus. After the application of the R_S rule, the algorithm will call all the constraints that have defined this identifier and the value of the P probability will be set for the rule with the R_D identifier. In the case that the R_E parameter is not defined, the probability is permanently set. In the case that the R_E identifier is specified, the value of the probability will be set until C derivations of the R_E rule will not be done. If the C parameter is not defined, the default value for C is set to one.

VI. Random Program Generation

Random generation of stimuli (programs) is based on our architecture of universal generation. We continue in our research in this direction. The difference from the previous version can be seen in the core of generator and in processing the specific inputs. [12] The architecture based on the grammar presented in this paper is shown in Fig. 2. The probabilistic context-free grammar is defined in the input structure called *Format*, while the constraints for rules are in input *Constraints*. The preprocessing (Preprocess) of inputs is the first step before the generation starts. Since context-free grammar cannot effectively define numerical ranges or names for jump instructions, we use the templating system Jinja2 [13] for the Python programming language [14] which allows us to define the cycles, branches and other special macros that we use. The demonstration of the *IMM* non-terminal definition for the derivation of random decimal number in the range from 1 to 1,000 through the library Jinja2 follows:

```
{% for i in range(1,1000) %}
    IMM → {{i}}
{% endfor %}
```

The output of preprocessing is an extended format of the probabilistic context-free grammar and constraints which already contains the complete definitions of the rewrite rules and constraints necessary to ensure the completeness and validity of the generated program.

The extended formats are processed by the core of the generator. It performs the application of the rules from the starting non-terminal with leftmost derivations. After the derivation of any rule is performed, the constraints for the relevant rule are triggered and thus the new probabilities are set for the given rules.

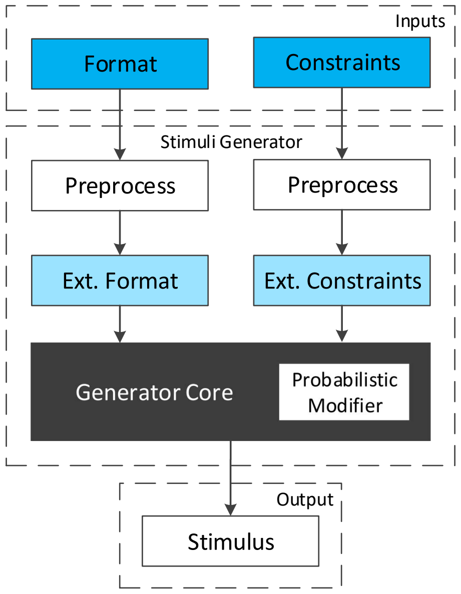

Fig. 2: The detailed architecture for a probabilistic context-free grammar based stimuli generation.

VII. Experimental Results

During our experiments, we have verified that the proposed method of encoding program instructions into the grammar is possible and is fully suitable for the generation of valid test programs.

We describe the experiment which is based on comparing the generation time of assembly programs with different number of instructions. Generation time is an important factor that affects the whole process of testing and verification of the system. It can significantly contribute to reducing the overall time needed for system testing. The comparison was done between this proposed approach of generation (referred to as USG generator), MicroGP tool (where we utilize only instruction library block), and our previously optimized generator of assembly programs for processors (referred to as RISC generator). We have defined adequate input structures for the creation of a valid assembly code for each tool. The results of our experiment can be seen in Fig. 3 and Table I.

The fastest tool is the RISC generator which is our specific generator, especially designed for RISC and VLIW processors. The generation time was less than 1 second for 25,000 valid instructions. The USG generator was in the second position with a generation time slightly over 1 second for the same number of instructions. The worst was MicroGP tool with 43 seconds for the same number of instructions. The generation speed of our generators is obvious. The main contribution which makes our approach different from conventional ones is in the description. We do not use any semantic information about the system. The whole process of the generation is based solely on ensuring defined constraints, without additional calculations and semantic dependencies. Thanks to this, we are able to save up to 42 seconds through the USG generator in comparison with the MicroGP tool.

TABLE I: The comparison of generation time for USG, RISC, and MicroGP tool (in seconds).

Number of instructions	1000	2000	5000	10000	15000	20000	25000
MicroGP	0.7	1.5	4.0	10.0	18.0	29.0	43.0
USG generator	0.1	0.1	0.2	0.5	0.7	0.9	1.1
RISC generator	0.1	0.1	0.2	0.3	0.5	0.6	0.7

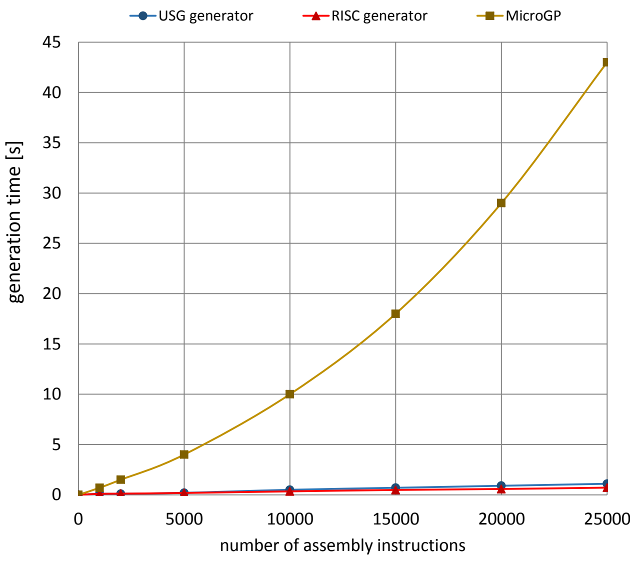

Fig. 3: The comparison of generation time for USG, RISC, and MicroGP tool.

However, there is also a minor difference between our approaches. The difference exists because grammar systems cannot effectively define the above mentioned numerical ranges or labels and all possible cases have to be enumerated. For this reason, a large set of rules is defined which results in browsing slowing down generation performance.

VIII. Conclusions and Future Research

In this paper, our research in the field of randomly generated test stimuli was presented and the application of the approach to a processor was described. In this case, stimuli representing the programs consist of instructions. We have demonstrated the universal architecture of stimuli generation which is based on two input structures. We have defined the format using a probabilistic context-free grammar which is a context-free grammar with added probabilities for rewrite rules. Through the constraints we ensured an application of rewrite rules in the defined grammar so that the final program was valid for the given processor. The experiment with generation time demonstrated a substantial acceleration against the conventional tool.

Although we presented our approach on the processor, the architecture allows us to generate stimuli for different systems which will be the focus of our future research. We shall also examine optimization possibilities of the complete generation process in order to achieve higher quality of the generated stimuli.

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, ARTEMIS JU under grant agreement no 621439 (ALMARVI) and BUT project FIT-S-17-3994.

References

[1] S. Roy and S. Ramesh, "Functional verification of system on chips - practices, issues and challenges," in *Proceedings of ASP-DAC 2002*, 2002, pp. 11–13.

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

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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

[10] 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

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

[12] 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

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

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