Universal Pseudo-random Generation of Assembler Codes for Processors

Ondrej Cekan, Marcela Simkova, Zdenek Kotasek
Faculty of Information Technology, Brno University of Technology
Bozetechnova 2, 612 66 Brno, Czech Republic
Tel.: +420 54114-{1361, 1362, 1223}
Email: {icekan, isimkova, kotasek}@fit.vutbr.cz

***Abstract*—The paper describes a universal generation of test stimuli based on solving constraints. The architecture of the universal generator consists of two formal models. The first one is used for describing the generated scenario and the second one for specifying constraints for this scenario. The generation of the assembler programs for Application-Specific Instruction-set Processors (ASIPs) is an example of the use of this architecture. The necessary steps needed to generate a valid assembler code are described. The quality of the generator is measured by the instruction and statement coverage in functional verification.**

I. Introduction

All the time, it is important to deliver faultless and high-quality systems to customers. When aiming at systems without errors, they must be properly tested and all the usual and unusual combinations of inputs that may arise must be taken into account. With the increasing complexity of the system, the complexity associated with testing the functionality of the system increases as well. Simple circuits can be tested manually, but for more complex circuits manual testing would be time consuming. Due to this fact, a technique called functional verification was developed [1]. Functional verification [2] compares the behaviour of DUT (Device Under Test) with the behaviour of its reference model described in another programming language. On the basis of comparing their outputs for the same inputs, the correctness of the system is evaluated with respect to the system specification and implementation. During functional verification, complete functionality of the system must be tested. For preparing test stimuli, a constraint-random generator is typically utilized. In addition, functional verification provides the so-called coverage report which lists the areas of the system that are and are not covered. The principle of functional verification is shown in Figure 1.

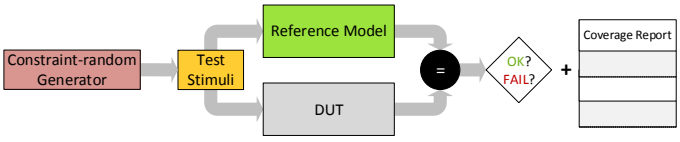

Fig. 1. The basic principle of functional verification.

A generator of test stimuli is important for functional verification. Generation of the test stimuli or more complex testing scenarios is needed for thorough testing and verifying the correctness of a system. As the test stimuli usually have different formats for different systems, a generator that is universal, parameterized and fast is needed. The reason is that very often, a new generator for every specific system is currently developed separately. There already exist generators that are parameterized [3] or they accept different formats of inputs for the generated data [4], but as far as we are informed, no test stimulus generation principles which are universal and simultaneously parameterized exist in the literature. In the following sections, we present our solution for a universal generator which is based on solving constraints [5] and uses two formal models whose generation process is very fast.

This paper is organized as follows: Section II describes the universal approach of generating test stimuli for many applications. In Section III we demonstrate the application of this approach for the automated generation of assembler programs for ASIPs. The results of experiments with the test stimulus generator are summarized in Section IV. Finally, Section V concludes the paper and outlines the direction of our future research.

II. Universal Generation of Test Stimuli

The architecture of the universal test stimulus generation consists of two formal models (see Figure 2): *description of the problem* and *constraints defined for this problem*. Each of these formal models is represented by one file with a specific purpose and format.

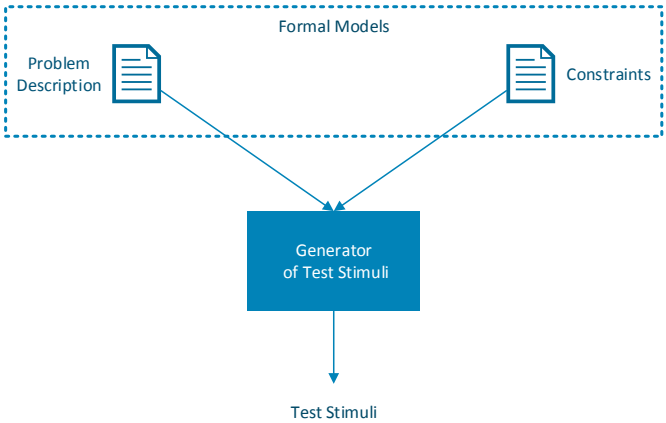

Fig. 2. The architecture of the test stimuli generation.

The *Problem Description* model contains information about what we want to generate.

The second model, called *Constraints*, contains restrictions and limitations for the problem described in The Problem

Description model. This model defines valid combinations, the ordering and conditions for test stimuli composed by the constraints.

The core of generation is the Generator of Test Stimuli which takes these two formal models (files) as inputs and generates test stimuli through their combined use. Theoretically, with this architecture, we are able to cover many areas. An important prerequisite is the creation of a set of general constraints that can be used directly or combined together when solving a variety of target generation problems. In this paper, we are creating a set of constraints that serve to generate an assembler code for ASIPs. Our previous work [6] shows another application of this universal generator, in particular, the generation of mazes for checking fault-tolerant qualities of the robot controller.

A. The Problem Description Model

The Problem Description model contains three basic parts for problem definition (see Figure 3): the substitute part, the variable part and the syntax part.

Fig. 3. The parts of the Problem Description model.

The *Syntax part* defines the syntactic strings one after another that we need to generate pseudo-randomly as test stimuli. In each syntactic string, a variable or a substitute can appear, but it must be defined in the Variable part or in the Substitute part. Variables and substitutes will be replaced in the syntactic strings. The Syntax part represents static values while the two remaining parts represent dynamic or changing values in the syntactic strings.

The *Substitute part* defines all possible substitutes which will be pseudo-randomly replacing any syntactic string defined in the Syntax part. The Substitute part is similar to the enumeration data type. In every new cycle of the generation process some replacement is taken pseudo-randomly for a given substitution.

The *Variable part* defines the variables in a general sense, where for each of them a value is assigned pseudo-randomly based on its data type. In every cycle of the generation process new values are assigned. Although this part is very similar to the *Substitute part*, it was divided for a better understanding of the difference in the use of both parts. In terms of implementation, it is also a crucial difference between the parts that contributed to the division.

B. The Constraints Model

As mentioned earlier, the constraints represent conditions and limitations for the generated test stimuli. The constraints ensure a valid test stimulus generation. The Constraints model is specific for each system as well as the Problem Description model, therefore, various restrictions are applied to different systems.

C. Generator of Test Stimuli

The generator of test stimuli explores combinations of syntaxes, substitutes, and variables so that all constraints are satisfied. The generator must be able to understand constraints which are applied to the Constraints model.

III. Assembler Test Stimulus Generation for Processors

The generation of assembler code for ASIPs is one example of the use of a universal generation concept presented in the previous section. We designed the Problem Description model and the Constraints model for processors. Afterwards, the whole generator was integrated into the Codasip Framework [7] of the Codasip company. The previous models were described in general, the models in this section are described specifically for a one test-case: the Codix RISC processor [8].

A. The Problem Description Model for Processors

The *Syntax part* defines strings that we want to generate. We want to generate an assembler code, so this part contains all instructions of the processor. Each defined instruction consists of an identifier and an instruction syntax. The identifier is used for links between the constraints. The instruction syntax is the body, where replacement will be carried out and then the modified instruction syntax will be printed. The example of one instruction is:

```
ori { "dst = or src1, imm" }
```

where *ori* is the identifier, the string between curly braces is the instruction syntax, *dst* and *src1* are substitutes and *imm* is a variable.

The *Substitute part* defines the set of strings to be replaced in the instruction syntax. This part is typical for the register definition. It is specified here which substitutes will be replaced by a specific string. The example of one substitute is:

```
dst { r0|r1|r2 }
```

where *dst* is a substitute string; r0, r1, and r2 are the replacements.

The *Variable part* defines the variables in a general sense. It is usually used for assigning a number into an immediate operand in the instruction syntax or for assigning a string into a label in the jump instructions. The example of one variable is:

```
VAR16 imm
```

where *VAR16* is the 16-bit integer number and *imm* is the name of the variable.

B. The Constraints Model for Processors

Because the instructions and operands in the assembler code line cannot be generated fully randomly, we introduced a method based on controlling the process of generating stimuli by using the constraints. The process is controlled by the constraints and all constraints must be fulfilled. We have

developed several constraints which solve typical problems of the assembler code generation. The set of constraints for the processors is shown in Figure 4. The successive application of the constraints is also demonstrated. The constraints with a star mark are evaluated only once during the generation, while other constraints are evaluated for each instruction. The description of the constraints and their typical application in the assembler code generation is shown in Table I.

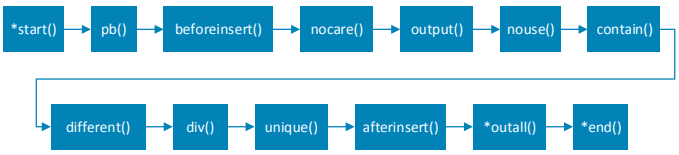

Fig. 4. The set of the constraints for generating the assembler code.

TABLE I. THE CONSTRAINTS FOR ASSEMBLER GENERATION.

Constraint	Description	Used for
*start()/ *end()	Generates an instruction as the first/last one.	regs initialization, halt generation
pb()	Sets probability of an instruction generation.	limits for instr.
beforeinsert()	Inserts instruction before a specific instruction.	latency maintain
nocare()	Sets a substitute that cannot carry the value.	conditional instr.
output()	Sets a substitute of any instruction as an output.	regs initialization
nouse()	A variable cannot be used in the next instruction.	latency maintain
contain()	A variable assigns a previously generated value.	jump instr., label
different()	A variable must be different from any variable.	jump instr.
div()	A value of variable must be divisible by a number.	mem aligned access
unique()	A value must be unique in the whole program.	jump instr., label
afterinsert()	Inserts an instruction after the specific instruction.	latency maintain
*outall()	At the end, prints instruction in the contain() link.	label of instr.

IV. EXPERIMENTAL RESULTS

The experiments were performed on processor Codix-RISC [8]. Codix-RISC is a 32-bit RISC processor with 32 general purpose registers, 512kB memory and 2729 instructions. For this processor, we have automatically generated the UVM(Universal Verification Methodology)-based functional verification environment and the verification process was running in the ModelSim simulator from Mentor Graphics [9].

In our experiments, we examined the instruction and the statement coverage for our programs in functional verification. Coverage is expressed in percentages. We have generated 1980 programs with 100 and 1000 instructions using the universal generator described before and then compared the results with the MiBench test program suite [10] that was used in the company as the main test suite. The MiBench suite is composed of 1980 programs with approximately 100-1000 instructions. We have investigated the maximal coverage and the number of programs that are included in the test suite. The results of our experiments are demonstrated in Fig. 5. The graph's x-axes are plotted in a logarithmic scale.

The maximal coverage of our experiments was 88.09% for the instruction coverage and 85.65% for the statement coverage. These values were achieved for programs which were generated by the proposed universal generator of test stimuli. For 100 instructions, more programs were necessary than for 1000 instructions. In the program, there are some initial sequences, therefore coverage of programs with 100 instructions grow slowly. In comparison to the MiBench, higher coverage was achieved, namely +14.29% for the instruction coverage and +1.97% for the statement coverage. Moreover, the overall coverage for our generator was achieved more quickly and was higher for any number of programs than the coverage achieved by the MiBench test suite.

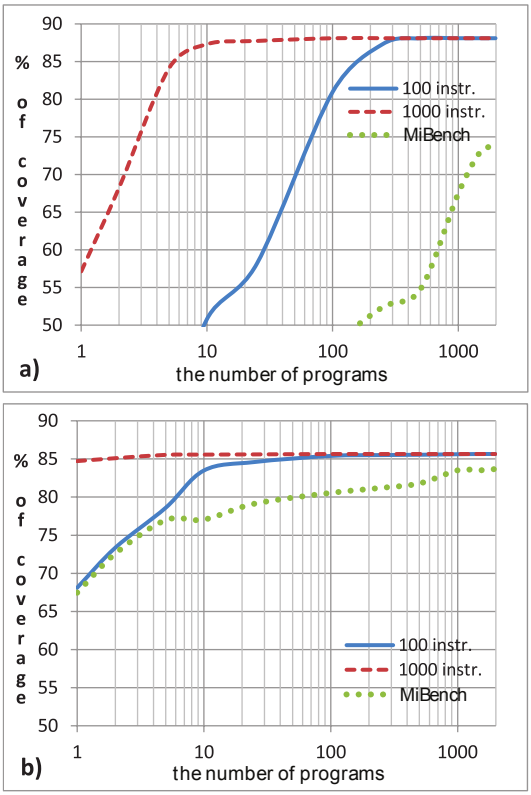

Fig. 5. Achieved a) instruction coverage and b) statement coverage in functional verification.

V. CONCLUSIONS AND FUTURE RESEARCH

In this paper, we proposed a universal pseudo-random test stimuli generation module which can be used in functional verification as an input test generator. We have integrated this generator into the Codasip Framework and demonstrated a working example on processor Codix RISC. We have measured coverage of the generated test suite of assembler programs in verification. The results were compared with a benchmark set of test stimuli that was available. Our generator provides better results in terms of coverage than the MiBench test suite. It should also be noted that our generator is very fast. It generates 1000 instructions per approximately 60 ms, depending on the PC used.

Our future work will concentrate on studying other possibilities of using this universal generator and on the creation of general sets of constraints so that it will be possible to generate various test stimuli for different scenarios.

ACKNOWLEDGMENT

This work was supported by the following projects: BUT project FIT-S-14-2297, National COST LD12036, project IT4Innovations Centre of Excellence (ED1. 1.00/02.0070), COST Action project "Manufacturable and Dependable Multicore Architectures at Nanoscale".

REFERENCES

[1] E. Zhang and E. Yogev, "Functional verification with completely self-checking tests," in *Verilog HDL Conference, 1997., IEEE International*, Mar 1997, pp. 2–9.

[2] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Des. Test*, vol. 18, no. 4, pp. 36–45, Jul. 2001. [Online]. Available: http://dx.doi.org/10.1109/54.936247

[3] Z. Michalewicz, K. Deb, M. Schmidt, and T. Stidsen, "Test-case generator for nonlinear continuous parameter optimization techniques," *Evolutionary Computation, IEEE Transactions on*, vol. 4, no. 3, pp. 197–215, Sep 2000.

[4] K. Ranganathan, M. Rangarajan, P. Alexander, and T. Regan, "Automated test vector generation from rosetta requirements," in *VHDL International Users Forum Fall Workshop, 2000. Proceedings*, 2000, pp. 51–58.

[5] L. Kotthoff, "Constraint solvers: An empirical evaluation of design decisions, circa preprint," in *CoRR, abs/1002.0134*, 2010.

[6] J. Podivinsky, O. Cekan, M. Simkova, and Z. Kotasek, "The evaluation platform for testing fault-tolerance methodologies in electro-mechanical applications," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*, Aug 2014, pp. 312–319.

[7] Codasip. (2013) Codasip - codasip framework. [Online]. Available: www.codasip.com/products/

[8] ——. (2013) Codasip - codix-risc. [Online]. Available: www.codasip.com/products/codix-risc/

[9] U. Hatnik and S. Altmann, "Using modelsim, matlab/simulink and ns for simulation of distributed systems," in *Parallel Computing in Electrical Engineering, 2004. PARELEC 2004. International Conference on*, Sept 2004, pp. 114–119.

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/WWC.2001.15