BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

PRINCIPLES OF TEST STIMULI GENERATION

PRINCIPY GENEROVÁNÍ TESTOVACÍCH STIMULŮ

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR **Ing. ONDŘEJ ČEKAN**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. ZDENĚK KOTÁSEK, CSc.**
ŠKOLITEL

BRNO 2021

Abstract

The research presented in this thesis is focused on the design of general principles in the field of generating stimuli for various systems. Stimuli represent the input data of a system that determines its behavior. A significant advantage is the use of these principles in the field of functional verification. Functional verification is one of the verification techniques that verifies the correct behavior of the system by monitoring its inputs and outputs. The proposal took into account four key criteria in terms of generating stimuli - parameterizability, speed, randomness, versatility.

Based on the design, the architecture of stimuli generation for general use was defined. Input structures are used to describe stimuli, which define the desired stimulus format and the constraints imposed on it. Thanks to this, it is possible to obtain both a valid stimulus and change the constraints during the generation, which is especially suitable for obtaining higher coverage in functional verification. The general definition of stimuli is ensured by a formal description. The research defined the principles of creating stimuli for processors, functional units and application data. The presented method achieves an improvement over conventional approaches.

Abstrakt

Výzkum prezentovaný v této práci je zaměřen na návrh obecných principů v oblasti generování stimulů pro různé systémy. Stimuly představují vstupní data systému, které určují jeho chování. Značnou výhodou je využití těchto principů v oblasti funkční verifikace. Funkční verifikace je jedna z verifikačních technik, která ověřuje správné chování systému monitorováním jeho vstupů a výstupů. Návrh zohlednil čtyři klíčová kritéria z hlediska generování stimulů - parametrizovatelnost, rychlost, náhodnost, univerzálnost.

Na základě návrhu byla definována architektura generování stimulů pro obecné použití. Pro popis stimulů slouží vstupní struktury, které definují požadovaný formát stimulu a omezující podmínky na něj kladené. Díky tomu je možno získat jak validní stimul, tak měnit omezující podmínky v průběhu generování, což je vhodné především pro získání vyššího pokrytí ve funkční verifikaci. Obecnost definice stimulů je zajištěna pomocí formálního popisu. V rámci výzkumu byly definovány principy tvorby stimulů pro procesory, funkční jednotky i aplikační data. Představený způsob dosahuje zlepšení oproti konvenčním přístupům.

Keywords

Verification Stimulus, Test Vector, Functional Verification, Constraints, Processor, Robot controller, Assembler, Maze, Formal Grammar, Probabilistic Constraint Grammar

Klíčová slova

Verifikační stimul, testovací vektor, funkční verifikace, omezující podmínky, procesor, řadič robota, assembler, bludiště, formální gramatika, pravděpodobnostní omezená gramatika

Reference

ČEKAN, Ondřej. *Principles of test stimuli generation.* Brno, 2021. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Zdeněk Kotásek, CSc.

Principles of test stimuli generation

Declaration

I declare that I have prepared this dissertation independently under the guidance of doc. Ing. Zdeněk Kotásek, CSc. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Ondřej Čekan
March 3, 2021

Acknowledgements

I would like to thank my supervisor, doc. Zdeněk Kotásek, for his professional guidance and valuable advice that led to the creation of this dissertation. I would also like to thank my colleagues from DCSY for factual comments and ideas. I also thank my wife and the whole family for their support.

Contents

Chapter 1

Introduction

Electronic circuits are becoming more and more at the forefront of people's daily lives, and it could be said that since the 21st century, the world cannot exist without them. Even young children encounter electronics built into their toys without realizing what is behind the magic sounds or songs. Micro-controllers, which care about controlling certain devices, are located where we probably would not expect them to be. They can be found in toys, kitchen appliances, smart pendants or even in jewelry [33]. Such use of micro-controllers represents the high-end of today, and the circuits that control such a thing are not abnormally reliable. Circuits are tested for basic functionality and there is a lack of deeper verification of correct behavior. Also, there is no reason to do exhaustive testing, because circuit verification is time consuming and manufacturers try to meet at least the basic things needed to sell a working product. On the other hand, there are applications in which a lack of design or functionality of the system can mean a serious risk and endanger human lives. Such applications are called safety-critical [59] and are represented by areas of the automotive, aerospace, space applications or medical industries.

The circuits of these systems must be properly tested and verified for the correctness to prevent unexpected and undesirable behavior that could cause damage. These systems also contain additional circuitry to serve as a backup in the event of a fault or to monitor proper behavior. Such systems are called fault-tolerant [29, 37] and can work correctly even if some of their components fail.

If we focus on systems that do not contain design or implementation errors that could cause incorrect behavior, then these systems must be thoroughly tested. The usual but also unusual combinations of input values that may occur in a given system have to be taken into account. As the complexity of the system continues to grow, so does the complexity associated with thorough verification of its accuracy [56]. Simple systems are not difficult to test manually. For more complex systems, manual testing is very time consuming. Also, the formal techniques developed so far for the formal verification of large-scale systems fail. For this reason, a technique called functional verification [17] was developed. This technique verifies the correctness of the systems on the basis of monitoring their input and output values.

The functional verification process typically involves several separate blocks that interact with each other to verify the circuit as efficiently as possible. One of these blocks is the generator of input stimuli, which is the subject of this dissertation thesis. The values that enter the verified system are most often called stimuli, but can also be referred to as transactions, test vectors, or tests.

Each system is in principle different and therefore requires specific input stimuli for its operation [4, 58]. The design of such a specific stimulus generator significantly extends the total time required to test and verify the system, so the aim of this work is to provide a general approach to stimulus generation with respect to the use in the functional verification process, re-usability, test quality and generation speed.

1.1 Goals of the thesis

Within the dissertation thesis, two main goals and their sub-goals were stated, which lead to the fulfillment of the topic of this work:

1. **Design and creation of a universal test stimulus generator based on solving a constraint satisfaction problem, which will be especially suitable for the use in functional verification.**

   The test stimulus generator must be parameterizable so that it can be used in functional verification. The generator is then able to process the constraints during the verification run and adapt the generated test stimuli to achieve a higher coverage of the system functions. Inputs for the generator will be obtained from specially designed structures, which will define the format of the generated test stimuli and the constraints that will be applied in the process of generating these stimuli.

   The sub-goals for this goal are as follows:

   (a) Design and implementation of a stimulus generator based on input descriptions defining the format and constraints for stimuli.

   (b) Design of input descriptions for generating the first use case - programs for processors. Verification of programs in functional verification.

   (c) Design of input descriptions for another use case.

2. **Design of a general and uniform description (language) of various systems, which can be used to describe all the conditions and relations necessary to generate a valid test stimulus. The result of this activity will be the creation of test generation methods for various systems.**

   The main point is to create sets of input descriptions for various digital circuits (processors, functional units, fault-tolerant units, etc.) that the designed generator will use to generate test stimuli. The proposed input descriptions will be generalized and based on them, a language will be defined for a uniform description of stimuli of different digital systems. From this description, the key principles of test generation for given types of circuits will be extracted.

   The sub-goals for this goal are as follows:

   (a) Generalization of generation and description of stimuli, creation of a framework.

   (b) Transcription of stimuli using grammar. Evaluation of stimuli in functional verification.

   (c) Definition of stimuli for other systems.

1.2 Organization of the Thesis

The dissertation is created as a collection of selected published papers of the author. The research contribution of this dissertation is contained in six papers attached at the end of the thesis in the original published form.

The content of this work is arranged as follows. Verification, especially functional verification, is introduced in Chapter 2, along with the most common techniques that functional verification uses to increase the automation of verification process. The position of the generator and its importance in functional verification is also shown in this chapter, where the currently used input test generators are discussed. Chapter 3 summarizes the achieved results of the dissertation thesis according to the set goals. Finally, Chapter 4 concludes the thesis.

The six publications on which this dissertation is based are attached at the very end of the thesis.

Chapter 2

State of the Art

This chapter summarizes the current state of the art, which is necessary to get acquainted with and which is followed up in the dissertation. It is mainly a description of functional verification and the constraints satisfaction problem on which the design of generating stimuli for various systems is based.

2.1 Verification of Digital Systems

Verification [35, 36] is a process consisting of certain steps aimed at verifying the correctness of a hardware design with its specification. Hardware design can be described at any stage of abstraction - from high-level description to the physical placement of components. The correctness of the design is verified with its specification, which can also be defined in various ways - from behavioral description to timing requirements and other descriptions. However, the verification does not deal with the validation of the specification, i.e. the system does the actual intended activity that the customer expects from it, although it helps to make it more precise. The quality of verification depends on how detailed and correct the system specification is described and how its design is developed. As the digital system can be quite complex, most verification techniques are unable to verify the complete description of the system. For this reason, it is appropriate to divide the system into simpler units, which are verified separately. In practice, it is then much more manageable to verify such large circuits.

The main purpose of verification is to detect as many errors in the system as possible, especially in the initial stages of digital system design, in order to verify the correct behavior, speed up system testing and, as a result, deliver a cheaper, safer and error-free product to the customer.

If we compare verification with simple testing or simulation, the verification proves that the system does not contain errors for any valid input values (verifies all states), while simulation and testing only allows to reproduce erroneous scenarios and thus debug errors for certain inputs. Therefore, simulation and testing do not prove the faultlessness of the system and also do not verify all possible states of the system [19].

Two basic approaches to verification used in digital systems are recognized. It is a formal verification and a functional verification. This dissertation thesis does not deal with formal verification. That is the reason why it is mentioned only marginally. The main attention is paid to functional verification, which is used to verify the quality of input stimuli.

2.1.1 Formal Verification

Formal verification [39, 54] uses mathematical methods to verify the correctness of a system. These mathematical methods help us to describe the system or a property of the system. The result of the formal verification process is a proof of correctness for the specified condition or an example that violates the given condition. Formal verification offers three basic methods:

- Model Checking - verifies the property of the system by examining its complete state space.
- Equivalence checking - verifies two system models interpreting the same specification for equivalence in their behavior.
- Theorem proving - verifies the system or its property using proofs of mathematical logic.

Although formal verification provides a clear answer to the verified condition in the examined system, a number of problems can arise. One of them is infinity, where the result cannot be determined in real time. Another problem may be the so-called state explosion problem, during which the memory for storing states may not be sufficient. This problem occurs mainly with the Model Checking method. The state space exploration in formal verification is shown in Figure 2.1.

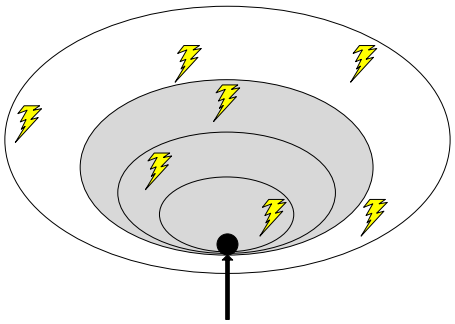


Figure 2.1: State space exploration in formal verification.

2.1.2 Functional Verification

Functional verification [46, 25] is a process in which the correctness of a system with respect to its specification is verified by setting its inputs and monitoring its outputs. Functional verification takes place in a simulation and therefore it does not provide evidence of system correctness. In contrast, it uses additional techniques, making the simulation itself more efficient. As shown in Figure 2.2, functional verification does not go through the state space systematically, one case after another as formal verification, but goes through it randomly in iterations, always from the initial setting, which is called *seed*.

Functional verification is based on two systems that are tested in parallel with the same input data. The first system is a hardware device described in the HDL (Hardware Description Language) language [5], referred to as DUT (Device Under Test) or DUV (Device Under Verification), which is verified for correctness with respect to its specification.

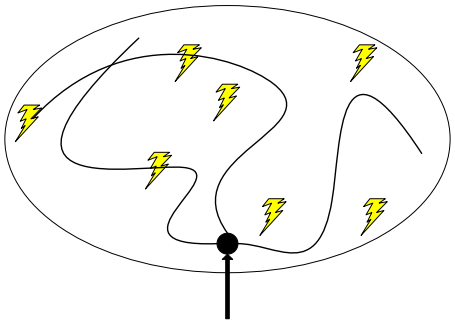


Figure 2.2: State space exploration in functional verification.

The second system is a model of the verified system that meets the same specification and is typically implemented in another programming language. The model is also typically implemented by another developer to avoid the same implementation bugs and the same problems in misunderstanding the specification. The model is often referred to as a *reference* or *golden* model. The same test stimulus is typically applied to the input of the two systems, which is typically obtained using a test stimuli generator. The outputs of these systems are compared for equality.

The output of functional verification is the result of a comparison with the outputs of both systems as well as information about code coverage or functional coverage [61, 23]. Code coverage [53] expresses how much hardware source program (HDL) has been executed for a given test benchmark or test suite. Coverage is calculated as a percentage. If a given benchmark has a large coverage, then a significant part of the program was executed during its verification, and therefore there is a greater chance of detecting hidden design or implementation errors. In contrast, if the achieved coverage is low, the program executed a negligible part of the code, and so the system was not thoroughly verified.

When evaluating code coverage, several metrics can be tracked that focus on different parts of the design description [45].

The total code coverage consists of the following 4 metrics:

1. Statement coverage
   - verifies that each program command has been executed.

2. Branch coverage
   - verifies that each branch of the program has been performed.

3. Expression coverage
   - verifies that each expression has acquired all its values.

4. Condition coverage
   - verifies that each condition has acquired a Boolean value of True or False.

Within the simulation environment, it is also possible to define user conditions that form the second group of coverage - functional coverage [16], to be monitored in the system. These conditions are typically specific cases of which the programmer is aware and which must not be omitted during verification. Examples are input data thresholds, certain

sequences of transitions between states, or other situations.

When verifying functional coverage, 2 metrics can be monitored:

1. Data-oriented coverage
   - verifies that a defined combination of data values has occurred.
2. Control-oriented overage
   - verifies whether a defined sequence of steps in the program has occurred.

If all defined conditions are met, 100% coverage of the digital system is achieved, which is a key criterion in terms of functional verification. The principle of functional verification described above is shown in the Figure 2.3.

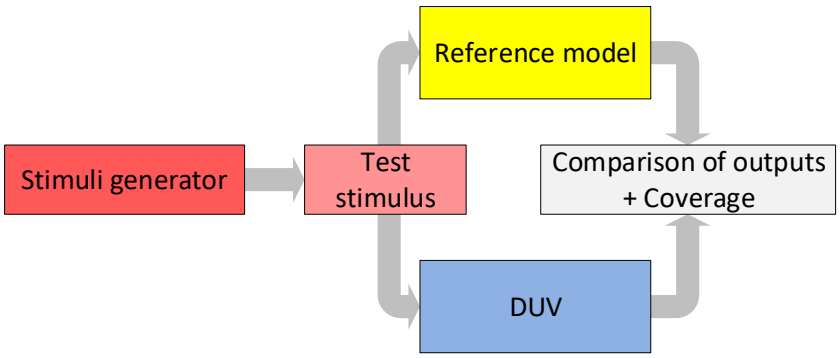


Figure 2.3: Principle of functional verification.

The goal of functional verification is to constantly increase coverage and thus achieve an ideal 100% coverage of the set system metrics. How coverage can be increased, preferably in an automated manner, is described in the following subchapter.

2.2 Principles of Functional Verification

This subchapter presents three basic principles of functional verification, which are currently used to verify the correct behavior of the system and achieve the highest possible coverage of its monitored metrics. These principles differ from each other mainly in the level of automation.

2.2.1 Random Stimuli Generation

The first principle of functional verification is the random stimuli generation [20] shown in Figure 2.4. This principle is the simplest of the presented principles with the smallest degree of automation. It uses a random stimulus generator that allows no configuration or control. The basic behavior of the system is verified on the generated tests. If some system metrics are still not verified (uncovered), the verification engineer manually generates a set of verification stimuli (direct tests) depending on the coverage analysis until it covers these metrics. The main disadvantage of this principle is the generation of a large number of stimuli that are invalid and which cannot be considered as right input sequences of the

system. Valid sequences are in most systems a subset of all possible input combinations. This approach lacks any management and the entire success of the verification depends on the knowledge and experience of the verification engineer. This approach is also very time consuming.

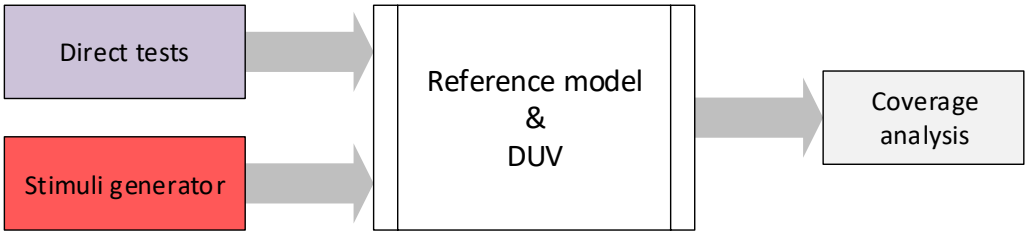


Figure 2.4: Principle of random stimuli generation.

2.2.2 Random Stimuli Generation Based on Constraints

The second principle extends the random stimuli generation by controlling using constraints [22]. The principle is shown in Figure 2.5. When verifying the system, only certain test scenarios are of interest because they are also valid. Using this principle, it is possible to generate specific and mainly valid stimuli that meet defined constraints. These constraints are created manually based on the system specification and represent inputs to the stimulus generator. Depend on them, valid verification stimuli are generated. However, some parts of the system may still remain uncovered, so additional constraints can be added manually to generate additional stimuli to cover these parts. Also, direct tests can be defined manually as in the previous principle. Verification of the system is accelerated in this case because invalid combinations are eliminated. The task of the verification engineer is facilitated and mainly focused on creating suitable constraints that will create the required stimuli.

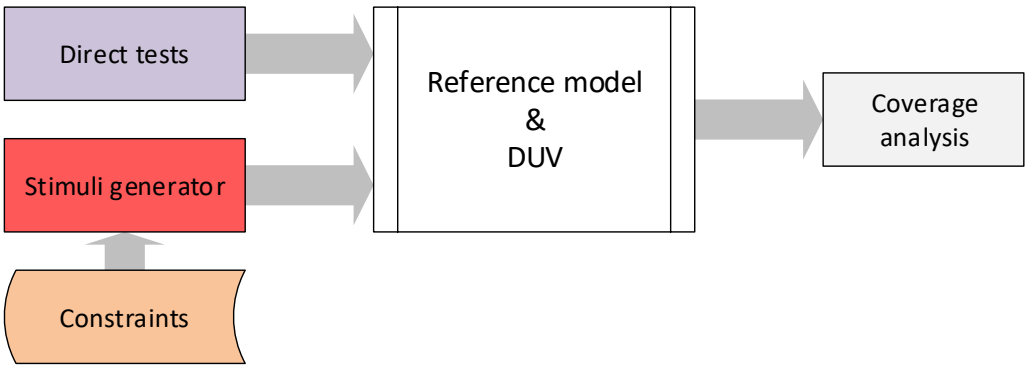


Figure 2.5: Principle of random stimuli generation based on constraints.

2.2.3 Coverage-driven Stimuli Generation

This last principle extends the previous principle of stimuli generation by controlling of coverage evaluation [18]. The principle is shown in Figure 2.6 and is characterized by

considerable automation in contrast to the previous principles. It uses the information available from the coverage analysis to automatically generate additional constraints in order to direct the next stimuli generation cycle and thus achieve higher coverage of system metrics. This principle is called coverage-driven verification. How the constraints are adjusted is not solved by the stimulus generator itself, but by higher logic, such as the genetic algorithm that performs this activity. This principle is characterized by minimal interaction with the verification engineer.

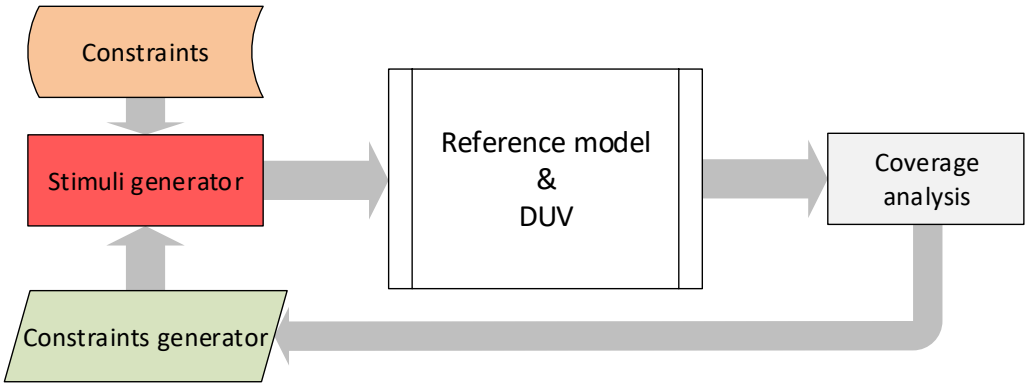


Figure 2.6: Principle of coverage-driven stimuli generation.

2.3 Constraint Satisfaction Problem

Constraint satisfaction problem (CSP) [20, 38, 41] is a general mathematical problem that is defined by a set of variables that can take values from a finite, nonempty, and discrete domain, and a set of constraints. Each constraint is defined above a certain subset of variables, for which it specifies valid values from a given domain that they can take. The result of solving the CSP is one or all assignments of values to the variables so that all constraints are fulfilled.

Definition 1 *Let X be a set of variables, D be a domain of values and C be a set of constraints. Then CSP is defined as a triplet (X,D,C) , where for each c ∈ C exists a pair (t,R), where t is an n-tuple of variables and R is an n-ary relation over D.*

CSP is an NP-complete problem, so for most complex problems there is no efficient algorithm for finding a solution [9]. When searching for the right solution, the algorithm must test all possible branches of the assignment, so it reaches a polynomial time complexity. In practice, the fulfillment of all constraints is often omitted for complex problems (e.g. some are not so necessary), but it is important to complete the calculation in real time. Solutions that have all the constraints fulfilled are called *exact*, otherwise they are called *approximate*.

If there exists such an assignment of all the variables from their domain of values that all the constraints are fulfilled, then such a solution is *satisfactory*. However, if the variables cannot be assigned so that the constraint is fulfilled, then the solution is *unsatisfactory*. When searching for the right solution, a request can be made to find:

- The exact solution.
- All exact solutions.
- Optimum exact solution.
- An approximate solution that satisfies the conditions for the defined variables.

Simple and typical examples of CSPs belong to the N queens problem [6] or the graph coloring problem [48]. These examples are shown in Figure 2.7.

The task of the N queens problem is to determine the position of individual N queens on a chessboard with dimensions NxN so that none of the queens are endangered. Queens represent variables, the dimensions of the chessboard represent the domain of values, and the requirement not to endanger queens is a constraint placed on them.

The situation is similar to the graph coloring problem. The requirement is to color the graph so that the two adjacent areas on the map have always different colors. Areas on the map represent variables, the available colors are domain values, and the constraint is in the form of a link between two adjacent areas. This problem can be converted into a so-called constraint graph, which is equivalent to a CSP.

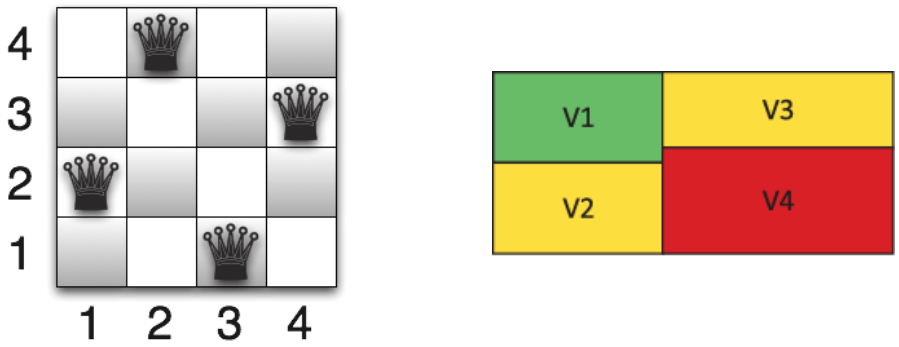


Figure 2.7: Example of the N queens problem (left) and the graph coloring problem (right).

CSP can be solved using several techniques. The most well-known of them is the backtracking technique. The backtracking [51] technique sequentially initializes variables and continuously validates any constraints placed on them. If a constraint is not fulfilled, the variable assignment is returned to the last valid instance that has an alternate assignment option.

Another technique is the constraints propagation [49]. This method is based on explicitly removing a value or combination of values from the domain, when searching for the correct assignment of variables. Gradually assigning values to variables will shrink the domain because a certain subset of constraints cannot be longer satisfied. In the domain of variables, there are always values that are available and that can be assigned without violating any constraints. If the variable no longer contains any value in its domain for assigning, then the solution does not exist.

Another option is to solve the CSP using the simplest technique called Generate-and-Test, which sequentially tests all the values assignments from the domain to the variables. If the constraints are satisfied for the tested combination of values, the valid solution was found.

2.3.1 Test Stimuli Generation Based on Constraints

As shown in the subchapter about functional verification, the test stimuli generator plays an important role in this process. Searching suitable stimuli can facilitate the whole process of circuit verification and thus speed up the development of the system and reduce its cost. The generation of test stimuli can be divided into two main categories - manual and automatic generation [63].

Manual generation is an engineering activity in the form of manual test creation. Verification engineers understand the design and structure of the verified system in detail, and therefore, they can focus on the baits of the system, its limit values, transitions between states, etc. Manual testing is a very time-consuming activity, so the results of which may not always be sufficient.

The second option is to use support programs that can create tests automatically. Such programs are based primarily on solving constraints, the solution of these programs is a valid test stimulus for the given system. A huge advantage of automatically generated tests is their randomness and speed. Thanks to these benefits, the system is verified to atypical and unpredictable combinations of input values that can occur in a given system. Also, the combinations, which do not usually arise during manual generation, are verified.

In practice, a combination of both stimuli generation approaches is used. The system is first tested on a small set of manually generated tests to determine basic functionality and after that it is fully verified on automatically generated stimuli as long as it is beneficial.

Test Stimuli Generator Requirements

The requirements for a test stimuli generator can vary in many cases, which is mainly due to the area of application of the generator. The topic of this dissertation thesis focuses on the use of a test stimuli generator in the field of functional verification, therefore all other areas of use will not be taken into account. In the field of functional verification, the requirements follow from the previous subchapters:

Parameterizability
: Parameterizability is an important feature for adjusting the behavior of the generator, which generates the desired output based on the entered parameters, thanks to which it adapts to the current needs of test stimuli to achieve high coverage. It is desirable that the generator inputs are separate so that they can be changed externally.

Speed
: The generator must be fast so that it does not slow down the already busy and time-consuming simulation of the verified system. The faster the generator is, the more inputs can be tested at the same time.

Randomness
: Random tests are a prerequisite for the uniform generation of all possible combinations of input values. This ensures to test unpredictable and marginal cases in the system. It is important that the probability of generating all parts of the stimulus is balanced.

Versatility
: Since each system is unique, it is suitable to design the generator in general so that there is no need to always create a new generator for a specific system and it is possible to use an existing stimuli generator for different systems. Again, this reduces the total time required to verify the system.

2.3.2 Current Research in the Field of Test Stimuli Generation

Current research in the field of test stimuli generation deals with the automatic generation of tests for one specific system, especially programs for the RISC (Reduced Instruction Set Computing) [15] type processor. The trend for many applications is to use either a universal processor or an Application Specific Instruction Set Processor (ASIP) due to its low cost, rapid development, acceptable performance and consumption. The required programs for processors are obtained from several specific input structures that describe a given processor at different levels. These input blocks are defined specifically for processors and therefore are not applicable to other digital systems or other application data.

The processor Instruction Set (ISA) [52] description is used as input, which is combined with another additional description that ensures the correctness of the generated program. The work [7] uses as a second description certain elements of the microarchitecture of the processor. The dissertation [31] from the Slovak University of Technology in Bratislava uses VHDL (VHSIC Hardware Description Language) [5] as the second description of the processor. Subsequently, it uses a genetic algorithm (GA) to modify the program so that the value of fitness or coverage in functional verification is as high as possible.

Another work [14], which automatically generates programs for processors, uses the representation of programs using acyclic graphs. The input of the generator is a hand-designed instruction library, which describes the assembler syntax of individual instructions and their valid combinations of operands. Together with predefined macros, processor description and real-time simulation, a valid program is generated that has sufficient coverage of processor functions. Here comes a more general description of the input stimuli for different processors, but the format and combination of all possible operands for individual instructions can be quite complex and extensive.

Test stimuli can also be generated using a Verification Description Language (VDL) such as SystemVerilog [32] associated with the ModelSim [27] simulation environment. Using this language, constraints can be defined and modeled for test stimuli represented by numbers. Modeling more complex stimuli is a problem as adjusting the generation during the process of functional verification.

Genetic programming techniques can also be used to generate programs for processors. The work [13] shows the generation of assembler programs based on the definition of instruction macros and their combination into a more complex whole (program). GA plays a key role in the selection of these macros and the assignment of their operands. The problem with this approach is mainly in the time-consuming task of compiling a valid program.

The current research in the field of test stimuli generation implies the need to design and create a universal test stimuli generator suitable for use in the principle of functional verification based on random stimuli generation driven by coverage.

An important condition for the current verification of the correct behavior of the system is the parameterization during the verification run for the possibility of modifying the properties of the generator and the generated stimuli. This makes verification as efficient as possible and it obtains the highest possible coverage of the code and key functions of the system. The effort is also to speed things up, so the universal generation principle should be able to easily create a test scenario. The reusability of already defined structures underlines the importance of universality and simplifies adaptation to a new field of application. A huge advantage can be its use both in generating stimuli for different hardware and application data for different software [44].

2.4 Formal Grammar

In the theory of formal languages, grammar [43, 34] is a mean of creating strings in a formal language. It consists of a set of production rules that determine how language strings can be created from a given alphabet. A string (also referred to a word) is defined as a finite sequence of alphabet symbols.

Definition 2 *An alphabet is a finite, nonempty set of elements that are called symbols.*

Definition 3 *Let Σ be an alphabet, then*

- *ϵ is a string over the alphabet Σ*
- *if x is a string over Σ and $a \in \Sigma$, then xa is a string over the alphabet Σ*

The language L over the alphabet Σ is defined as a set of strings from the closure Σ^* ($L \subseteq \Sigma^*$). A language can be finite - it contains a finite number of strings, or infinite - it contains an infinite number of strings. Any language always contains at least two elements - an empty set and an empty string ϵ.

The grammar generates the language while the automaton [30] receives it. The generation of a string in a formal grammar takes place from a single initial symbol, to which production rules are gradually applied in any order [40]. The production rules are applied as long as the string is not composed only of terminal symbols. The terminal symbol is a character from the alphabet Σ, which can appear in the generated string and which cannot be further changed by production rules. A string change using a rule is called a derivation.

According to Chomsky's hierarchy of languages, defined by Noam Chomsky in 1956 [11], grammars are divided into four types:

Type-0 grammars - unrestricted grammars

Unrestricted grammars include all formal grammars and can be recognized by the Turing machine. This is the grammar with the greatest expressive power. It is the quaternion:

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

N	is a finite set of nonterminal symbols (nonterminals).
T	is a finite set of terminal symbols, applies $N \cap T = \emptyset$.
P	is a finite set of production rules in the form: $(N \cup T)^* N (N \cup T)^* \rightarrow (N \cup T)^*$.
S	is the start symbol, $S \in N$.

Type-0 grammars are the family of all *Recursively Enumerable* (RE) languages.

Type-1 grammars - context grammars

Context grammars are formal grammars whose form of production rules preserves the context in which the word can be further rewritten. Grammar is the quaternion:

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

- N is a finite set of nonterminal symbols (nonterminals).
- T is a finite set of terminal symbols, applies $N \cap T = \emptyset$.
- P is a finite set of production rules in the form: $\alpha A \beta \rightarrow \alpha \gamma \beta$.
- S is the start symbol, $S \in N$.

The following applies to grammar production rules, $\alpha, \beta \in (N \cup T)^*$, $A \in N$, $\gamma \in (N \cup T)^+$. Type-1 grammars are the family of all *Context-Sensitive* (CS) languages.

Type-2 grammars - context-free grammars

Context-free grammars are special cases of context grammars, where the symbols α and β are omitted or replaced by the empty strings ϵ. For this reason, any nonterminal can be rewritten regardless of the surrounding context. It is the quaternion:

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

- N is a finite set of nonterminal symbols (nonterminals).
- T is a finite set of terminal symbols, applies $N \cap T = \emptyset$.
- P is a finite set of production rules in the form: $N \rightarrow (N \cup T)^*$.
- S is the start symbol, $S \in N$.

Type-2 grammars are the family of all *Context-Free* (CF) languages.

Type-3 grammars - regular grammars

Regular grammars are formal grammars with the smallest expressive power. Grammar is the quaternion:

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

- N is a finite set of nonterminal symbols (nonterminals).
- T is a finite set of terminal symbols, applies $N \cap T = \emptyset$.
- P is a finite set of production rules in the form: $A \rightarrow xB$ or $A \rightarrow x$, where $x \in T$ and $A, B \in N$.
- S is the start symbol, $S \in N$.

The grammar defined in this way is called the right regular grammar. If the grammar has production rules of the form $A \rightarrow Bx$ or $A \rightarrow x$, then it is called the left regular grammar. Type-3 grammars are the family of all *Regular* (REG) languages.

The relationship of the presented individual grammars according to the Chomsky hierarchy is expressed as $REG \subset CF \subset CS \subset RE$ and is shown in Figure 2.8.

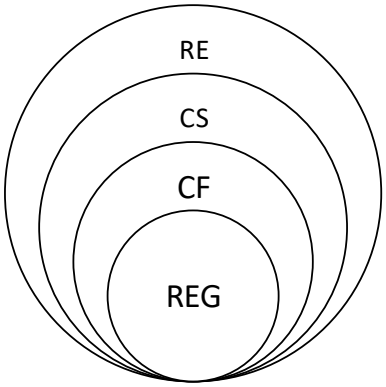


Figure 2.8: Chomsky's hierarchy of languages.

Chapter 3

Summary of Research Activities

This chapter summarizes the research results on which this dissertation thesis is based. The subchapter 3.1 discusses in details the individual steps of research that have lead to the fulfillment of the set goals of this thesis. A list of the six articles [I, II, III, IV, V, VI] included in this work, together with their abstracts, can be found in the subchapter 3.2. These articles form the basis of the thesis and are the output of individual partial steps of the author. The subchapter 3.3 encompasses the list of other publications of the author that are not part of this thesis, but should not be left out. The last subchapter 3.4 contains additional information on participation in projects and grants.

3.1 Research Process

The main idea of the whole research was to create a uniform and as general as possible a way of designing and creating stimuli, which can be applied to the widest possible set of different systems. This will reduce the time required to develop a specific stimuli generator for a particular system, speed up the whole process of verifying the correct behavior of the system and improve the achieved results. An important condition is the suitability of using such an approach in the process of functional verification. Based on these conditions and the set goals of this thesis, the first proposal of a universal method of generation was created, which was further developed and generalized.

The design of a universal method of test stimuli generation extends the classical functional verification scheme by two additional blocks, which represent the input to the stimuli generator itself based on the solving of constraint satisfaction problem (see Figure 3.1). The task of these blocks is to describe the various systems in a uniform way so that the generator itself uses only these blocks for its operation to generate valid tests and therefore does not require any further interventions in its source code. These input blocks provide the parameterization, i.e. the behavior of the generation process can be changed during the functional verification. It may be advantageous to generate these input descriptions (or at least one) automatically from source files for describing digital systems. The meaning of these blocks is discussed in details in the following subchapter 3.1.1 which is devoted to the overall architecture of universal generation.

For the proposed method of stimuli generation, the first use case was chosen - processor verification and the associated generation of assembler programs, which represent the required test stimuli. For these purposes, an available processor of the type RISC and VLIW (Very Long Instruction Word) was chosen. Based on the instruction sets of individual

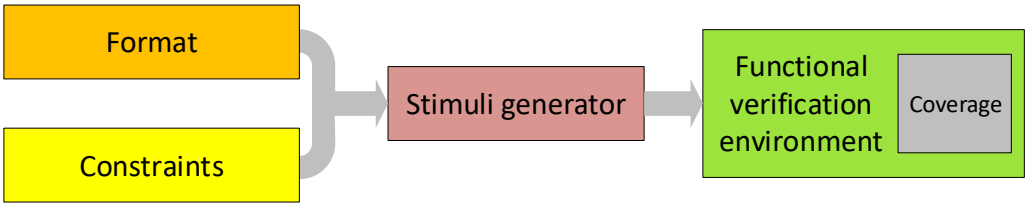


Figure 3.1: Design of a solution for stimuli generation based on input structures.

processors, input descriptions for generating valid programs were defined. An important process in the design of stimuli was the creation of a set of constraints that ensure a valid combination of consecutive program instructions while maintaining the greatest possible randomness. In total, 18 types of constraints were created, which can ensure the construction of a valid program. The effectiveness of this solution was verified in the functional verification of the selected processor. The knowledge from the construction of assembler programs was further used in the verification of a new approach in evaluating the software fault tolerance of memory elements of the processor to faults. Individual processor instructions were generated in triplicate, including security using both time and space redundancies. Using this method, it was possible to ensure the error-free execution of the processor program in the event of single fault and partly also multiple faults in the memory elements.

The following work was focused on the control unit of a robot searching for a path in a maze. It is an electromechanical system for verifying the effect of faults on both the electronic and mechanical parts. The robot control unit is implemented in the FPGA, while the mechanical robot itself is simulated in a simulation environment on a PC. The correct behavior of the robot is verified in a functional verification. The input of the robot, among other parameters, is a maze in the form of a bitmap image, in which the robot searches for a path from point A to point B. In this case, the maze is understood as stimuli. The behavior of the robot, in case of its failure, needs to be verified on various sets of mazes, which were also generated in this thesis, using the presented generation approach. Whether the generated maze sufficiently covered the state space of the robot was verified by functional verification. The most suitable dimension of the maze, the organization of the corridors and the initial and target position of the robot were taken into account. To generate all these factors, additional constraints were defined to ensure a valid maze creation.

The individual specific input descriptions of stimuli were subsequently generalized (bottom-up method, see Figure 3.2) using a formal grammar that is able to generate a specific language. The grammar design was based on the already existing probabilistic context-free grammar. It adds to the classic context-free grammar the probability with which the production rules are applied when rewriting a sentence form. This grammar alone would not be able to define the language needed to generate valid stimuli, so its form has been extended to include constraints. This grammar was designed in this thesis and is called a probabilistic constrained grammar, which has a greater expressive power than context-free languages. It allows to maintain the context (a certain state during generation) necessary for the construction of valid stimuli. The specific constraints created for specific systems in the previous author steps have been translated and encoded into a new grammar (uniform formal description). This removed unwanted system-specific links. The whole generation process was also accelerated and the resulting stimuli were improved.

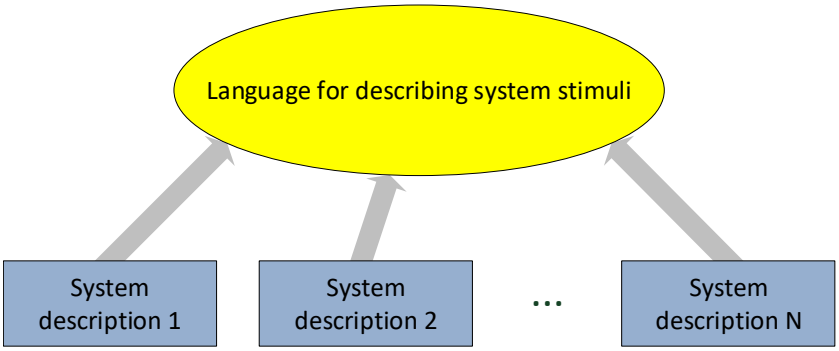


Figure 3.2: Generalization using the bottom-up method.

The last part of the research was devoted to the verification of this newly created grammar on the already investigated systems and the description of stimuli for another system (arithmetic-logic unit [57] and stepper motor [2]). Based on the findings, the principles and conditions necessary to ensure the generation of a valid test stimuli are described for individual selected systems.

3.1.1 Architecture of Universal Stimuli Generation

The principle of universal generation, which is shown in Figure 3.3, aims to simplify and speed up the generation of test stimuli for different systems.

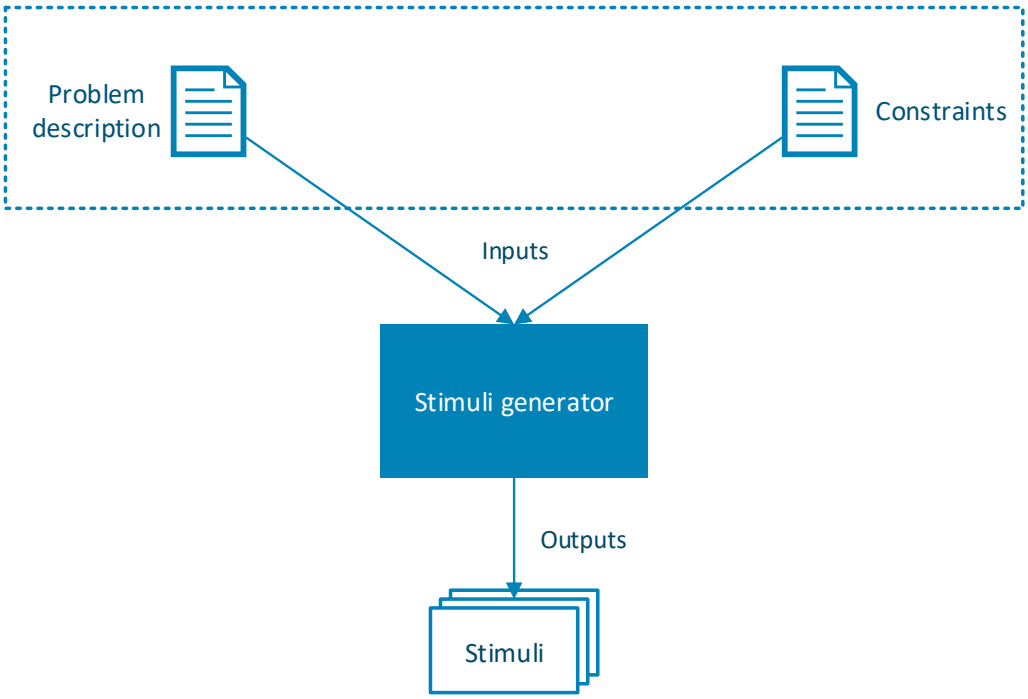


Figure 3.3: The principle of universal stimuli generation.

The main idea is based on two input descriptions (structures) that define the format of the generated data and the conditions determining how this data should be composed

during the generation. The description defining the format of the generated data is marked as the *Problem Description*, and the description defining the conditions and restrictions is marked as *Constraints*. These two descriptions represent the input to the generator itself, which generates valid stimuli on its output based on these descriptions. The obtained stimuli can then be transmitted to a specific system as input data.

When creating custom test scenarios, the user does not program any code, but only specifies the required format of the scenarios and the constraints. The generator then solves these definitions as the constraint satisfaction problem. An important prerequisite for this described generation principle is a wide set of input alphabets, which must generally describe different problems and constraints for the test scenarios.

This principle of stimulus generation is parameterizable. It can process and change constraints at runtime by modifying the input descriptions, and is therefore suitable for use in a functional verification process.

The generation process combines the problem definition so that all constraints are satisfied. The output of the generator is a sequence which corresponds to a defined problem and which forms the resulting stimulus.

3.1.2 Specific Structures and Processor Verification

The *Problem description* is the first input of the stimuli generator. To define the problem, i.e. what has to be generated, three basic parts have been designed (see Figure 3.4): Replacement, Variables and Syntax. The individual parts are defined in their own proprietary language.

Figure 3.4: Basic parts of *Problem description*.

The *Replacement* section defines the identifiers and all possible substitutions for which the identifiers can be replaced. This is similar to an enumerated data type in programming languages. Identifier is replaced by a specific string from a defined set of values. In each new generation cycle, a particular substitution is randomly selected for a given identifier. Substitution is used where it is necessary to substitute specific words or phrases in a certain generated string.

The *Variables* section defines variables in a general sense. A random value is assigned to each variable depending on its data type. In each new generation cycle, a new random value is assigned to the variable, if the variable exists in generated sentence.

The *Syntax* section syntactically describes the strings, one by one, to be randomly generated at the output of the generator as part of the stimulus. Identifiers from *Replacement* and *Variables* can be found in each defined syntax. These identifiers are replaced by a specific or random value, depending on the type of identifier. If no identifiers are found in the syntax string, their definitions are ignored and nothing is replaced. The Syntax section represents the static values in the generated string, while the other two parts represent the dynamic (changing) values in the generated string.

Constraints are the second input of the generator and represent the conditions imposed on the generated stimuli. By limiting possible solutions, the generation of valid scenarios for the selected system is ensured. Constraints are mainly conditions for data values (a variable can only take values from a certain range) or dependency conditions (some combination of values cannot occur after the currently generated combination). Like the Problem description, Constraints are unique to each system, and therefore different restrictions are applied to different systems. Constraints are also defined using their own proprietary language, and their syntax is similar to calling a function with parameters without a return value.

Generating Programs and Verification of the RISC Processor

The proposed stimuli generation architecture was implemented and input descriptions were defined for generating Assembler code for the processor type RISC and VLIW. To design the descriptions and test their functionality, the processors from Codix family [12] from Codasip company were used, namely Codix-RISC and Codix-VLIW processors.

To generate assembler code, the *Syntax* section contains the instruction set of the selected processor. Each defined instruction in this section consists of an identifier and its assembler representation. The identifier serves as a reference between the instruction and the constraint. The prepared assembler representations of instructions contain additional identifiers that are used in section Replacement (these are mainly registers) and Variables (these are mainly random numbers and strings).

Common instructions, which do not require special preprocessing, value restrictions, or cooperation with other instructions, do not need to define any constraints. However, some instructions require constraints to ensure their valid sequence and their valid operand values. Therefore, several constraints have been created that address typical problems in generating assembly code - jumps and calls, memory access, latency, program termination, labels, and more. There are in total 18 of these constraints for the Codix-RISC processor and they are demonstrated in Figure 3.5 for clarity. The figure also shows the sequential application of individual constraints. Constraints marked with an asterisk are evaluated only once during the entire generation process, other constraints are evaluated for each instruction in each generation cycle.

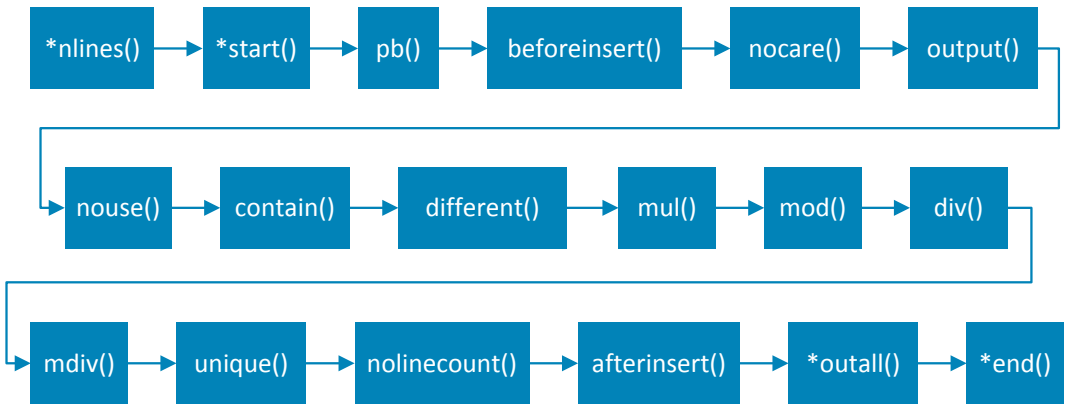


Figure 3.5: Constraints needed to generate assembler programs.

For the selected Codix-RISC processor, a verification environment created according to the Universal Verification Methodology (UVM) methodology [3] was available, which was used for experimentation in conjunction with the ModelSim simulator from Mentor Graphics [28]. The aim of the experiments was to determine the maximum coverage of key functions of the processor, which guarantees the compliance of the processor with its specification. As a part of the experiments, the coverage of processor instructions and states were evaluated. Using the proposed stimuli generator, 1980 programs of 100 and 1000 instructions were generated. All this was further compared with a set of test programs obtained from the MiBench tool [24]. The result of the comparison is shown in the Figure 3.6.

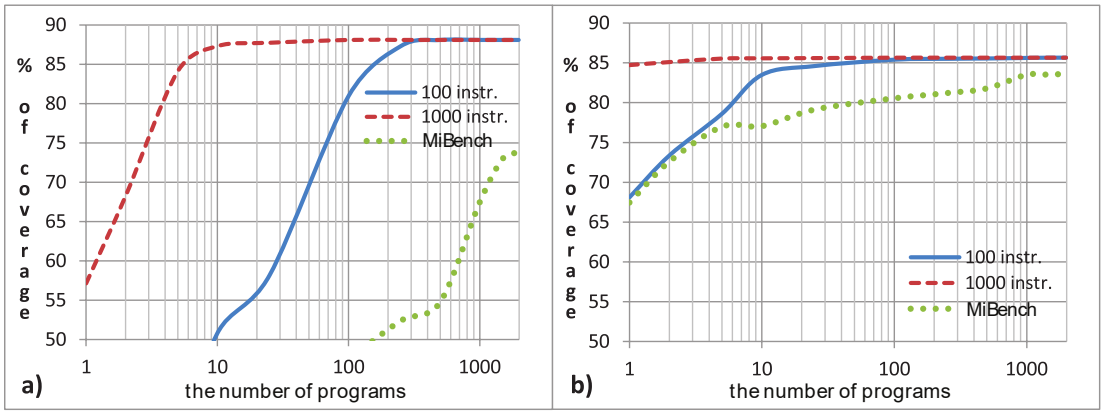


Figure 3.6: Achieved coverage of a) instructions and b) states in functional verification of the processor.

Programs generated using the proposed approach achieved higher coverage and in less time than programs from the MiBench test suite.

3.1.3 Evaluation of Software Fault Tolerance

This research was focused on evaluating the effect of transient faults caused by SEU (Single-Event Upset) faults [50]. These can occur completely unpredictably through charged particles or electromagnetic interference [10]. To deal with these faults, additional hardware or software techniques can be used. One option is to use software fault tolerance. The research used software redundancy, which was applied for securing data in memory elements. The processor with its registers and memory served as a sample device.

To solve this problem, an already designed and implemented stimuli generator was used. Its input structures were modified to generate the required and secured program. For software redundancy purposes, the new technique called Tripple Instructional Redundancy (TIR) has been proposed, which is similar to the Tripple Modular Redundancy (TMR) hardware technique [60], but runs at the software level. Figure 3.7 shows the technique of tripling instructions, including evaluating the majority using additional instructions that have the function of a voter. The condition for TIR must be that each of the instructions I1-I3 or the voters V1-V3 must use another part of the memory space that is disjunct with the others. Thanks to this, this method is able not only to detect the occurrence of a fault, but also to correct it. During the generation, a fault was also artificially inserted into the program in order to evaluate the proposed security methodology. A specific example of the assembler code along with a simulated SEU fault injection is also shown in the figure.

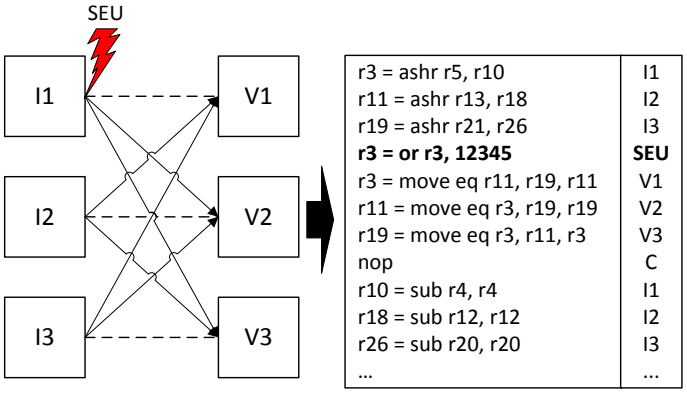


Figure 3.7: TIR software redundancy principle with example.

When defining the TIR in the universal stimuli generator, the definition of each processor instruction that works with the register or memory was tripled, and the newly created trio of instructions was followed by comparative instructions that have the function of a voter. To ensure the required separation of memory space between individual instances of triple instructions, it was necessary to define new constraints that will ensure the correct distribution of registers and memory addresses. The process of generating a program begins with selecting a random instruction. All relevant constraints for this instruction are evaluated, and in case of success, six instructions are printed to the output. Various instructions on the output create a fault-tolerant processor program. The SEU injection is simulated in the program by an instruction that is not secured by TIR and thus purposefully damages the value stored in the register/memory.

Within the experiments, two random programs were created in one generation cycle, each with 100 instructions. The first program was unsecured using TIR and no fault was injected to it. This program served as a reference. The second program was identical to the first, but was secured by TIR and a fault was simulated. The contents of the registers and memory after the execution of programs were compared to verify the effect of security on the proper behavior of the system. Up to 13 faults were injected into random parts of the secured program. The measured statistical data are plotted in Figure 3.8. Single faults were corrected in 100% of cases, double and triple faults in 95% and 90%, respectively. With another multiplicity of faults, the success of the repair was decreased sharply.

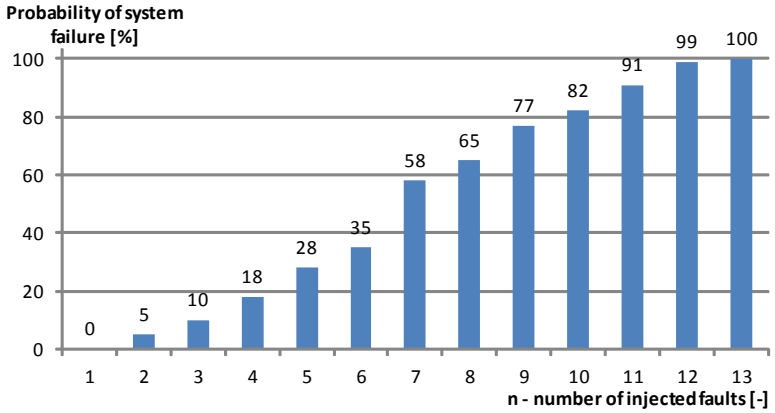


Figure 3.8: Probability of system failure in case of injection n faults.

3.1.4 Verification of the Robot Controller in the Maze

The robot controller in the maze is the second use case for which verification stimuli have been generated. This is a partial part of the work, which fits into the whole concept of the research group Diagnostics - the verification of methodologies on the electro-mechanical system. The robot controller (implemented in VHDL (VHSIC Hardware Description Language)) serves as a sample example available to the group. The task of the robot is to search the path from point A to point B. The robot has a maze stored in its memory, in which it orients itself using four sensors. The maze is one of several robot inputs that have to be processed.

The task of this research was to verify the robot controller in order to eliminate its design and implementation faults. This will ensure for subsequent future experiments with fault injection that the controller misconduct was actually caused by the fault injection and not the faulty design. For this reason, it was necessary to verify the robot's behavior on a sufficient sample of mazes to achieve the maximum possible coverage (preferably 100%) and a sufficient number of steps, so that any failure could manifest on its mechanical part.

A number of algorithms for generating a random maze exists, however, their construction is not so straightforward for encoding into the input structures of the generator. For this reason, an algorithm based on a binary tree was chosen and modified to minimize the dependency constraints of individual maze lines on previous ones. The maze representation is binary, where a value of 1 represents a wall and a value of 0 represents a corridor. This representation was later converted to a bitmap image. The maze creation algorithm requires that the two adjacent sides of the maze always have a corridor along their entire length. Thanks to this, it is then possible to identify the remaining cells of the maze so that there is always a path between any two points in the maze. Figure 3.9 shows the basic matrix of the maze, which was converted to a binary tree and new dependency conditions of individual rows were defined. Thus, if cells A or C were randomly selected for the corridor, then cell B or D will be the wall, and vice versa.

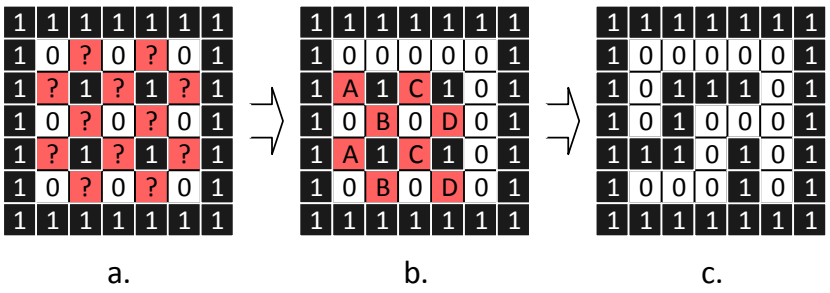


Figure 3.9: Conversion of the basic matrix of the maze (a) for the needs of the generator (b) and example (c).

The output of this work is a robot control unit without implementation errors, the list of used verification scenarios and the achieved coverage. For the experiments, 3 types of mazes were selected (see Figure 3.10), which differed in size and average number of steps - 7x7 cells (16 steps), 15x15 cells (93 steps) and 31x31 cells (433 steps). A total of 1500 verification scenarios were performed, the result of which is shown in Figure 3.11. A maximum coverage of 91.85% was achieved. This is mainly due to the „other“ branches in the controller implementation, which never execute (which is correct). As a suitable size of the

maze, the size of 15x15 cells was chosen, which has a sufficient number of steps and the maximum possible coverage.

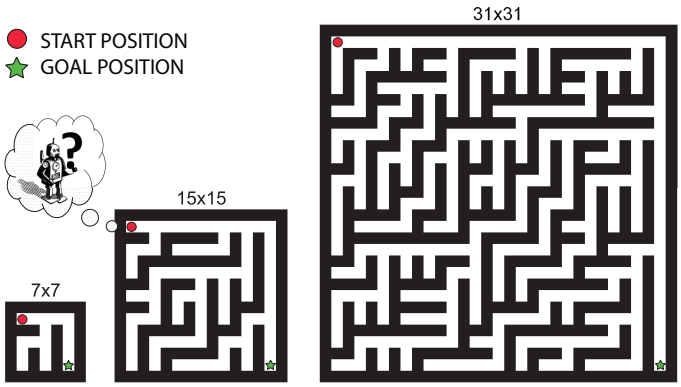


Figure 3.10: Three verified dimensions of mazes.

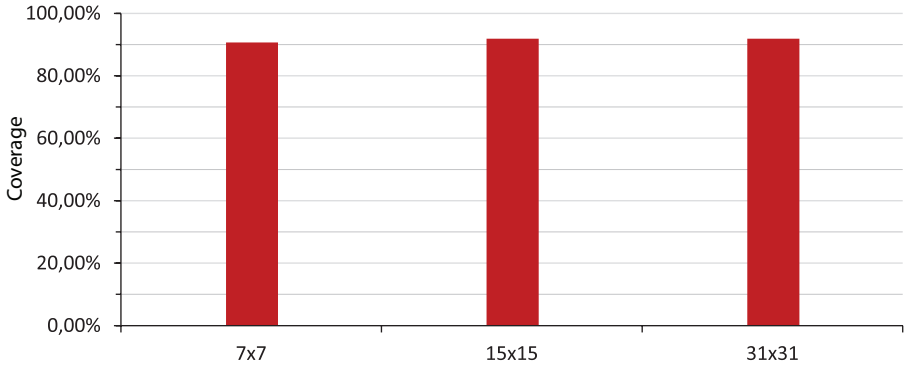


Figure 3.11: Achieved total coverage for each dimension of the maze.

3.1.5 Generalization Using Formal Grammar

Formal grammar is a means of representing any language. For this reason, it is appropriate to use or design a grammar that will be suitable to generate general stimuli. The aim of this research is to generalize the specific structures of the previously designed stimuli generator using a formal grammar based on the findings so far on specific test cases.

The newly proposed grammar is based on the already existing context-free grammar. Specifically, it is a probabilistic context-free grammar [21]. It is the quintuplet:

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

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

For the probability in the context-free grammar, the following Definition 4 must be applied.

Definition 4 *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 production rules* $r_1{:}A \to \alpha_1$, $r_2{:}A \to \alpha_2$, ..., $r_k{:}A \to \alpha_r$ *the following rule must be applied:* $\sum_{i=1}^{k} \pi_{ri} = 1$.

Thus, each production rule has assigned a probability with which the rule is applied at a derivation (the derivation means a transcription of a string by using the rule). The general probabilistic context-free grammars work first with the training dataset from which a probability of each rule is determined. Then, the given grammar uses this probability setting to generate strings.

For the purposes of this thesis (stimuli generation), three modifications of the general probabilistic context-free grammar were introduced:

1. Production rules may not be strictly defined with probability values in which they can be applied. If the probability is missing for a rule, it is calculated as the difference of 100% - $\sum$ defined_probability for given nonterminal. If there is no definition of probability for more than one rule, the probability for each of these rules will be the same and will be calculated as (100% - $\sum$ defined_probabilities) / number_of rules_without_probability for the given nonterminal. This defines the uniqueness for the application of rules without strictly specified probabilities. Probabilities will not be defined in most cases, because it is desirable to have the same probability for almost every rule due to the large number of different combinations.

2. Probabilities will not be determined from the training dataset, but will be determined by the developer based on the knowledge of the system. The aim is to limit the generation of certain elements, whose excess would be of no use in the resulting stimulus. If we consider the generation of processor instructions, then it is advisable to limit the generation of, for example, a group of jump instructions. The excess of instructions of this group would cause little usability of the code, because most instructions would be skipped.

3. Production rules, which have some additional constraints, must be clearly identifiable - they must have an identifier. The assigned probabilities of grammar rules will change during the generation of stimuli, according to the constraints defined in the following text of the thesis. Therefore, it is necessary to have a clear identification of the rules. Rules will be denoted by a combination of alphanumeric characters.

The probabilistic context-free grammar defined in this way was used to design a new grammar, which the author of this thesis calls Probabilistic Constrained Grammar. This new grammar is a pair:

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

H is a probabilistic context-free grammar.

C is an ordered list of constraints for grammar H.

Constraints limit the grammar in the application of production rules for a given nonterminal, and therefore limit the set of all possible strings in a given formal language. For the set of production rules R, where the nonterminal X is on the left side of the rule applies that the right side of the production rules represents the domain of values for the given nonterminal X. The constraints were designed as quintuple with respect to the needs of stimuli:

`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 whose application removes the constraints.
C (optional) is the number of derivatives of the R$_E$ rule before the constraint is removed.

The task of the constraint is to set the probabilities during the generation so that the result is a valid stimulus. After applying the R$_S$ rule, all constraints that have this identifier defined are invoked and the probability value P is set to the rule with the identifier R$_D$. If the parameter R$_E$ is not defined, this probability is set permanently. If the identifier R$_E$ is specified, the probability value will be set until the C derivation of the rule R$_E$. If C is not specified, the default value of C is set to one derivation.

Probabilistic context-free grammar itself defines a static language, which is given by a fixed probability setting, however, its behavior, in conjunction with constraints, is dynamic and the generated language changes during the application of production rules.

Generation Process

Random stimuli generation is based on the introduced universal generation architecture and the introduced probabilistic constrained grammar. The difference compared to the original version is in the core of the generator and the processing of specific inputs. The architecture is shown in Figure 3.12. Probabilistic context-free grammar is defined in the input marked *Format*, while restrictions for individual grammar production rules in the input *Constraints*. The first step before the generation is the preprocessing of the inputs. Since grammars cannot effectively define, for example numeric ranges, a template system [55] was used that allows the definition of cycles, branches, and other special macros suitable for this purpose.

The output of the preprocessing is the *Extended format* and *Extended constraints*, which already contain a complete definition of the production rules necessary to ensure the validity and completeness of the generated program. It is sufficient to regenerate these extended input descriptions only when changing the original inputs, otherwise it works directly with this format and is no longer generated again.

The extended input descriptions form the resulting probabilistic constrained grammar and are further processed by the generator core itself. It applies the production rules from the initial nonterminal S using the leftmost derivations. After applying any rule, the constraints that using these rules are invoked and the probabilities with which they can

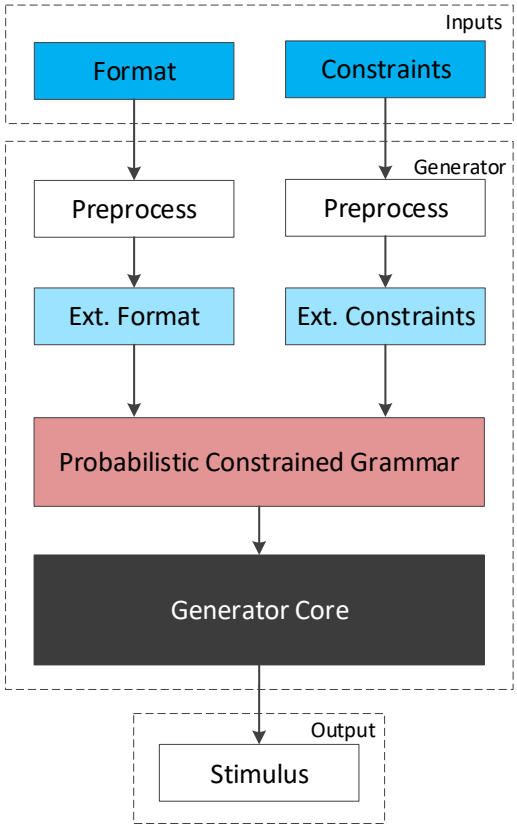


Figure 3.12: Detailed architecture of universal stimuli generation based on probabilistic constrained grammar.

be applied are adjusted. This directs another iteration of the grammar derivation, which ensures the validity of the generated stimulus.

3.1.6 Principles of Creating Stimuli for Various Systems Using the Grammar

This subchapter shows sample examples, which explain the principles of creating stimuli not only for systems that were introduced in this work, but also for other systems of similar or different nature. Probabilistic constrained grammar is a general way of stimuli definition, therefore the success of creating a grammar for a particular system always depends on the developer. It is important, how effectively he is able to encode the problem into production rules. The following subchapters show the solutions for the sample systems and their necessary key conditions that must be fulfilled in order to generate a valid stimulus.

Assembler for RISC Processor

Processor instructions need to be divided into several groups according to the type of instruction. Each group is defined by its own nonterminal, to which there is a path through the application of production rules from the start nonterminal. Each group can have an adjusted probability value depending on the type and number of instructions in the group so that the distribution of instructions in the program is as even as possible, and thus there are no more instructions than others. Jump instructions have typically less probability than,

for example, arithmetic ones, because their excess would cause low usability of program instructions. Each created group is then further divided into subgroups defined by another nonterminal, which combine instructions with the same format. For example, instructions that work with two registers are in a different group than instructions that work with a register and a value. The nonterminal of a subgroup is then replaced by an expression containing both nonterminal and terminal symbols. Nonterminal symbols are replaced by specific instruction names and operand values, which create one specific stimulus instruction at the output.

However, such a grammar would never end up generation process without constraints and the processor itself would not be able to handle the instructions. The generated program would lack the appropriate labels, it would also not ensure correct and aligned memory access and would not maintain the necessary latency to store the result in memory and its subsequent use.

Example: Creating a grammar for a RISC processor from Codasip. Let us assume that the nonterminal S is the starting symbol of the grammar. The processor instruction set, which represents about 60 instructions, can be divided into 5 groups of instructions - arithmetic (ARITHM), memory (MEMORY), conditional (CONDIT), jump (JUMPS) and others (OTHER). For these groups, it is possible to define production rules with modified probabilities subsequently (abbreviated notation - character '|' indicates 'or'):

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

If these production rules are followed by the start nonterminal, cyclic instruction generation is achieved. The group of arithmetic instructions (ARITHM) can be further divided into another 6 subgroups with different formatting, which determines a specific nonterminal. This includes a subset of instructions using two registers as operands (ARITHM_RR), register and value (ARITHM_RI), three registers as operands (ARITHM_THREE), sign extension instructions (ARITHM_EXT), assignment instructions (ARITHM_ASS) and instructions for loading a value into the upper half of the operand (ARITHM_LUI). Rules defining these groups derive nonterminals already on the specific syntax of the given instruction as they have defined in their instruction word. Specific registers and numerical values are still hidden behind other nonterminals. An example of definitions of such rules is as follows:

```
ARITHM → ARITHM_RR|ARITHM_RI|ARITHM_THREE|ARITHM_EXT|ARITHM_ASS|ARITH_LUI

ARITHM_RR → DST = ARITHM_NAME SRC, SRC EOL
ARITHM_RI → DST = ARITHM_NAME SRC, IMM EOL
ARITHM_NAME → add|sub|add|or|xor|shl|shr
dst0: DST → r0
src0: SRC → r0
dst1: DST → r1
src1: SRC → r1
dst2: DST → r2
src2: SRC → r2
...
jmp: JUMP → jump NAME EOL ARITHM NAME: EOL
```

```
nj1: NAME &→ str1
nj2: NAME &→ str2
eol: EOL → \n
```

It may seem that the nonterminals *DST* and *SRC* representing the registers are identical, because as a result there is a specific register behind them. That is the reason why we do not need to have two different definitions of rules. In reality, however, it is necessary to distinguish these operands for a given instruction, because for rules deriving the nonterminal *DST* there are different constraints than for the nonterminal *SRC*. Due to maintaining latency between instructions, i.e. for certain instructions, it is not possible to use the result of the previous operation immediately in the next tact, because the result is not saved yet.

The nonterminals *DST* and *SRC* are defined in a similar way and already contain specific processor registers. The nonterminal ARITHM_NAME is replaced to the specific name of the arithmetic operation of the processor. The whole branch of the derivation tree for the arithmetic sum operation is shown in Figure 3.13. It is obvious that a simple modification can also generate a binary representation of the program.

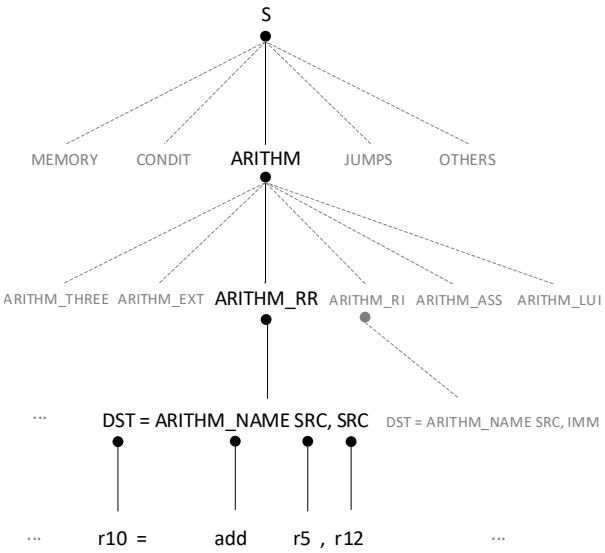


Figure 3.13: Derivation tree for arithmetic sum operation.

The constraints for such grammar can be defined as follows, to ensure the correctness of the instructions in the program.

If the arithmetic operations have a latency of 2, i.e. the result stored in the register under the nonterminal *DST* cannot be used in the next tact as the source register for all other processor instructions. Therefore constraints, the activation of which causes that the transcription of the destination register invalidates (sets the probability to zero) the corresponding rule for the same source register in the one subsequent instruction, has to be defined in the following way:

```
cons(dst0, src0, 0, eol, 1)
cons(dst1, src1, 0, eol, 1)
```

```
cons(dst2, src2, 0, eol, 1)
```

For jump instructions, only forward jumps are considered, because the specific values of the registers are not known at the time the program is generated, and thus an undesired endless loop would occur when returning in the program. The jump instruction *jump NAME* will be generated by using the *jmp* rule, including its label marked by *NAME*: nonterminal. Between the jump instruction and its label, any other instruction can be placed, except of 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* nonterminals 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. The necessary condition is that there must be a sufficient number of these labels in the grammar.

```
cons(nj1, nj1, 0)
cons(nj2, nj2, 0)
```

The end condition for generation is the achievement of a certain number of instructions. For this reason, it is necessary to restrict the grammar so that after the application of a certain number of new lines (identifier *eol*) the generation is terminated. It means that the output string does not contain nonterminals. This is accomplished by adding the following sequence of production rules:

```
start: S → START
end: START → nop(100%)
START → ARITHM START

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

Immediately after applying the first *start* rule, the constraint is invoked that prevents the *end* rule from being applied to the next 1000 instructions (new lines), which sets the required number of instructions in the stimulus. When this number is reached, the probability for the *end* rule returns to 100% and the cyclic generation of instructions ends with the final instruction *nop*.

Maze in Bitmap Image Format

To generate a maze in the form of a bitmap image, the encoding format of the .bmp image must be known. It is an uncompressed format that is stored pixel by pixel. The pixel value is stored in a certain number of bits, the length of which determines the possible color palette. The structure of a bitmap image can be divided into three mandatory parts - BMP header, DIP header and image data. The BMP header contains basic information for identifying the image. The DIP header describes the stored image itself, how the stored image data should be accessed. In the case of generating random mazes with a fixed size (e.g. 7x7 cells), it is possible to set BMP and DIP headers statically to constant values. Otherwise,

it is possible to use tools for pre-processing the input structures of the generator. For the purposes of this work, image data are stored in hexadecimal as a trio of words representing the *RGB* color palette (red-green-blue). Each cell of maze is represented as 8x8 pixels for the needs of the robot control unit. Overall, it is necessary to generate 56x56 pixels. The wall of the maze is defined by black color, while the corridor by white color.

The introductory part of the grammar design for maze generation includes the definition of production rules that address the bitmap image format. There are definitions of static headers and colors that are used in rules that generate image data. The production rules for color definition are already prepared to generate 7 pixels in a row. The maze must always have an odd number of cells so that there are no irregular corridors or walls. These production rules take the following form:

```
start: S → BMPHEAD DIBHEAD PIXDATA

BLACK → 0x00 0x00 0x00
WHITE → 0xFF 0xFF 0xFF
BLACK8 → BLACK BLACK BLACK BLACK BLACK BLACK BLACK BLACK
WHITE8 → WHITE WHITE WHITE WHITE WHITE WHITE WHITE WHITE
```

It is not necessary to show specific rules for replacing the nonterminals of the *BMPHEAD* and *DIBHEAD* headers to explain the principle. The image data itself are hidden under the nonterminal *PIXDATA*, which generates individual lines of pixels. Since the cell has 8 pixels, 8 identical rows are always generated. A necessary condition for generating a continuous maze, is the split of rows into even and odd, as was introduced in the proposed algorithm earlier. This is because an even row is always affected by its previous odd row, which sets values for this even row based on a random selection of a corridor or wall.

For the robot control unit, it is necessary that the maze is always closed and there is a continuous path from any cell to another, therefore a wall is always generated around the maze. In addition, at least two adjacent sides of the maze must be a corridor, so the top and right sides were chosen. These conditions are already taken into account when designing production rules, which have the following form, including special macros that facilitate writing:

```
PIXDATA → {% for i in range(1,8) %}
            BLACK8 BLACK8 BLACK8 BLACK8 BLACK8 BLACK8 BLACK8
          {% endfor %}
          {% for i in range(1,8) %}
            BLACK8 WHITE8 WHITE8 WHITE8 WHITE8 WHITE8 BLACK8
          {% endfor %}
          ODD
          {% for i in range(1,8) %}
            BLACK8 BLACK8 BLACK8 BLACK8 BLACK8 BLACK8 BLACK8
          {% endfor %}

odd: ODD → ODDBODY EVEN
end: ODD → (100%)
EVEN → EVENBODY ODD
```

```
ODDBODY → {% for i in range(1,8) %}
            BLACK8 A BLACK8 C BLACK8 WHITE8 BLACK8
          {% endfor %}

EVENBODY → {% for i in range(1,8) %}
            BLACK8 WHITE8 B WHITE8 D WHITE8 BLACK8
          {% endfor %}

Ab: A → BLACK8
Aw: A → WHITE8
Bb: B → BLACK8
Bw: B → WHITE8
Cb: C → BLACK8
Cw: C → WHITE8
Db: D → BLACK8
Dw: D → WHITE8
```

The ending condition is similarly defined as for the processor, i.e. specific constraint is invoked after the start rule, which in case of reaching the required number of lines with pixels, sets the transcription to the empty string (epsilon) for the rule where nonterminal *ODD* is on the left side. The remaining constraints set an odd line based on an even line:

```
cons(start,end,0,odd,2);

cons(Ab,Bb,0,Bw);
cons(Aw,Bw,0,Bb);
cons(Cb,Db,0,Dw);
cons(Cw,Dw,0,Db);
```

Input and Output Operands for Arithmetic-Logic Unit

The arithmetic-logic unit (ALU) is another example for which a grammar was designed to obtain stimuli. However, it is not just about generating inputs for this circuit, as in previous works, but with the help of the expressive power of probabilistic constrained grammar, it is also possible to describe the expected output, which can be part of the resulting stimulus. This makes it possible to quickly verify the correctness of the output when verifying the circuit. Similarly, in the case of functional verification, it is possible to exclude a reference model and thus compare the output of the verified circuit directly with the value of the stimulus. This method can also be used for other circuits, e.g. for the calculation of the cyclic redundancy check (CRC) [47].

The ALU typically has two input operands A and B, which are n bits long. Another input is the selected operation, which in this case is set to addition with carry. Output R is the result of an operation over the input operands.

The process of creating grammar production rules can be divided into several sections - Inputs, Logic and Result. Each section contains its own appropriate production rules. The most complex section is the *Logic* section, in which it is necessary to ensure that the

context of individual subtotals is correct and propagated into the final result for the given operation. A schematic representation of these sections is shown in the Figure 3.14.

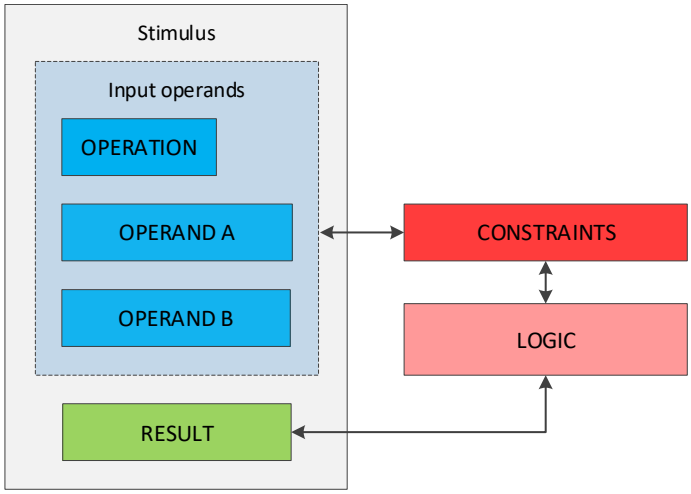


Figure 3.14: Schematic representation of grammar definition for ALU.

The resulting stimulus is made up of four lines, which correspond to random inputs and the expected result. Values are represented in the binary numeral system. The first line is the selected operation addition with carry, the second and third lines contain the operands, and the last line is the result of the operation over the operands. The bit widths of the input values can be chosen arbitrarily according to the required circuit. In this case, a width of 1 bit per operation and 8 bits per operands and result were chosen for clarity. The figure also shows constraints because they form an integral part of the proposed grammar.

The grammar production rules are designed so that each operand is divided into N other unique nonterminals, where N is the number of bits per operand. The same applies to the definition of the result. This division is very important for constraints, which will ensure that the correct result is determined. Each of these new nonterminals for the individual bits of operand A can then take the value zero or one. This achieves a randomly generated number in the first operand. In the case of generating bits of operand B, it is already needed to keep in nonterminals the value that operand A acquired for the positionally same bit. That is, each bit of operand A could be zero, one, zero with carry bit, one with carry bit. This fact must be taken into consideration when creating rules for operand B and create other nonterminals for it *BxA0*, *BxA1*, *BxA0C*, *BxA1C*. The definition of these rules is shown here:

```
A -> A7 A6 A5 A4 A3 A2 A1 A0
B -> B7 B6 B5 B4 B3 B2 B1 B0

A7 -> '0'|'1'
A6 -> '0'|'1'
...
A0 -> '0'|'1'

B7 -> B7A0|B7A1|B7A0C|B7A1C
B7A0|B7A1|B7A0C|B7A1C -> '0'|'1'
...
```

```
B0 -> B0A0|B0A1|B0A0C|B0A1C
B0A0|B0A1|B0A0C|B0A1C -> '0'|'1'

R -> R7 R6 R5 R4 R3 R2 R1 R0
R8|R7|R6|R5|R4|R3|R2|R1|R0 -> '0'|'1'
```

Based on the definition of these production rules, it is possible to determine the truth table of addition with carry (see Tab. 3.1).

Table 3.1: Truth table of addition with carry.

Ai	Bi	Ri	Ci+1
Ai->'0'	BiA0->'0'	Ri->'0'	0
Ai->'0'	BiA0->'1'	Ri->'1'	0
Ai->'0'	BiA0C->'0'	Ri->'1'	0
Ai->'0'	BiA0C->'1'	Ri->'0'	1
Ai->'1'	BiA1->'0'	Ri->'1'	0
Ai->'1'	BiA1->'1'	Ri->'0'	1
Ai->'1'	BiA1C->'0'	Ri->'0'	1
Ai->'1'	BiA1C->'1'	Ri->'1'	1

Without constraints, the generation of individual operands for the rules proposed in this way would be completely random, so it is necessary to create a set of constraints that will respect the truth table. Based on the generated value (zero or one) for individual bits of operand A, specific constraints will be activated and will limit the use of rules for operand B, i.e. set the correct context:

```
cons(A0->'0', B0->B0A1, 0);
cons(A0->'0', B0->B0A1C, 0);
cons(A0->'1', B0->B0A0, 0);
cons(A0->'1', B0->B0A0C, 0);
...
```

After performing these constraints, each bit of operand B always has only two rules that can be used to generate its value. Depending on which rule is randomly used, the value of the corresponding result bit is set using additional constraints. At the same time, the possible carry bit is set for the next bit. The constraints that provide these actions are briefly shown here:

```
cons(B0A0->'0', R0->'0', 100);
cons(B0A0->'0', B1->B1A0C, 0);
cons(B0A0->'0', B1->B1A1C, 0);
cons(B0A0->'1', R0->'1', 100);
cons(B0A0->'1', B1->B1A0C, 0);
cons(B0A0->'1', B1->B1A1C, 0);

cons(B0A1->'0', R0->'1', 100);
```

```
cons(B0A1->'0', B1->B1A0C, 0);
cons(B0A1->'0', B1->B1A1C, 0);
cons(B0A1->'1', R0->'0', 100);
cons(B0A1->'1', B1->B1A0, 0);
cons(B0A1->'1', B1->B1A1, 0);
...
```

This process of creating production rules and constraints is suitable for other arithmetic and bitwise operations. An important prerequisite is to design additional nonterminals so that the necessary context is maintained using appropriate production rules.

Stepper Motor Signals

Verifying the correct behavior of the stepper motor is another area of interest. The stepper motor is being investigated as a drive for smart electronic locks, which must be secured both from the point of view of cyber security and from the point of view of fault tolerance. During the functional verification of the stepper motor, it is again necessary to generate input stimuli (signals), according to which the angle of rotation of the stepper motor will be evaluated. Based on the angle of rotation of the motor in the event of a fault, it is examined whether the smart lock has been unlocked unauthorizedly or has not been locked during the locking sequence.

Input stimuli are designed directly for the stepper motor itself, because there was no control unit available to verify the real injection of faults. In fact, the missing control unit and the fault injector can be replaced by the stimuli generator. Thanks to this, it is possible to find out how the motor behaves with variously changed values on its input pins.

Input stimuli are generated for the available stepper motor model, which is part of the MATLAB application and the Simulink package [42, 2]. This model requires at its input a voltage for its 4 coils (0V or 5V) changing over time - stepping. For this reason, the individual steps in the stimulus are represented by lines. Each line consists of five values, where the first value represents the timestamp in which the step occurs. The remaining four values represent the voltage levels on the four coils (COIL1 - COIL4) of the stepper motor, which ensure its rotation. Logic levels are used to define the voltage (logic 0 = 0V, logic 1 = 5V). The resulting format of one motor step is as follows:

```
TIMESTAMP,COIL1,COIL2,COIL3,COIL4
```

Many steps must be defined (tens to thousands) to turn the stepper motor around its entire axis one or more times. Several steps obtained during the generation process define one input stimulus. For this step format, it is necessary to create a probabilistic constrained grammar. For a valid continuous motor rotation in one direction (or damaging such input by a fault), it is necessary to keep a valid sequence of stepper motor coils. In the case of a 4-phase stepper motor, 8 steps are required. If it is a stepper motor with a gearbox, many more steps are required.

During the creation of the grammar, several variables were defined using preprocessing templates, which could be used to modify the stimulus generation parameters. Thanks to this, it was possible to apply and test various scenarios of stepper motor attacks. These are 4 parameters:

time - maximum timestamp value (end of generation).
minstep - minimum time between steps.
maxstep - maximum time between steps.
pcoil - probability of logical 0 or 1 for each coil.

When generating a valid sequence of 8 steps of a stepper motor, it is necessary to keep the state in which the generation is located and on the basis of it determine the following values for individual motor coils. For a valid sequence of steps, coils 1-4 are always activated individually or in combination with an adjacent coil (COIL12, COIL23, COIL34 or COIL41). Preservation of the context in which of the 8 steps the generation is located is achieved by production rules and constraints, which set the values of the other coils when generating the value for COIL1. This achieves the cyclic generation of 8 steps and the rotation of the stepper motor in one direction. These rules can look like this:

```
STEP -> TIMESTAMP ',' C1 ',' C2 ',' C3 ',' C4
c11: C1 -> C11
c12: C1 -> C12
c12|...|c18: C1 -> C13|C14|C15|C16|C17|C18
C2 -> ZERO|ONE
C3 -> ZERO|ONE
C4 -> ZERO|ONE

C11|...|C18 -> ZERO|ONE
ZERO -> '0'
ONE -> '1'
```

The constraints provide control of the generation so that when using the rule *c11* for step 1 of 8, the probabilities of the rules for nonterminals C11, C2, C3 and C4 are set, whereby the generated values will acquire a valid sequence of step 1,0,0,0. At the same time, the following derivation is forced for the nonterminal C1 using the rule *c12*.

The value of the timestamp (nonterminal *TIMESTAMP*) will be determined by other production rules, which are pre-generated according to the specified time interval for individual steps. After preprocessing, these rules look like this:

```
t0: TIMESTAMP -> T0
t1: TIMESTAMP -> T1(0%)
t2|...|tx: TIMESTAMP -> T2(0%)|...|TX(0%)
T0 -> '0.0'
T1 -> '0.5'
T2 -> '1.0'
...
TX -> '60.0'
```

By supplementing the grammar with constraints, the generation of valid steps will be ensured. For each step, the timestamps will be selected sequentially from T0, T1 and so on by creating constraints, that during the application of the *t0* rule will set the use of the *t1* rule in the next derivation and similarly.

An example of constraints to ensure valid sequence of steps is shown here:

```
cons(t0,t1,100,t1); // timestamp
cons(t1,t2,100,t2);
...

cons(c11,c12,100); // step sequence
cons(c11,c11,0);
cons(c12,c13,100);
cons(c12,c12,0);
...

cons(c11,C11->ONE,100); // coil values in a given step
cons(c11,C2->ONE,0);
cons(c11,C3->ONE,0);
cons(c11,C4->ONE,0);

cons(c12,C12->ONE,100);
cons(c12,C2->ONE,100);
cons(c12,C3->ONE,0);
cons(c12,C4->ONE,0);
...
```

If values other than only 0% and 100% are used in constraints to set the correct sequence of steps, a certain percentage error can be entered into the generation of individual coil values and a fault at the input of the stepper motor can be simulated.

3.2 List of Publications Included in the Thesis

This subchapter shows an overview of published articles on which this dissertation thesis is based, including their abstracts. The full texts of all these articles can be found at the end of this thesis in the chapter Related Papers.

3.2.1 Paper I

PODIVÍNSKÝ Jakub, ČEKAN Ondřej, ŠIMKOVÁ Marcela and KOTÁSEK Zdeněk. The Evaluation Platform for Testing Fault-Tolerance Methodologies in Electro-mechanical Applications. *Microprocessors and Microsystems*, vol. 39, no. 8, 2015, pp. 1215-1230. ISSN 0141-9331.

Author participation: 35% | Citations: 2 | Impact factor: 1.045 (Q3)

Abstract

The aim of this paper is to present a new platform for estimating the fault-tolerance quality of electro-mechanical applications based on FPGAs. We demonstrate one working example of such EM application that was evaluated using our platform: the mechanical robot and its

electronic controller in an FPGA. Different building blocks of the electronic robot controller allow to model different effects of faults on the whole mission of the robot (searching a path in a maze). In the experiments, the mechanical robot is simulated in the simulation environment, where the effects of faults injected into its controller can be seen. In this way, it is possible to differentiate between the fault that causes the failure of the system and the fault that only decreases the performance. Further extensions of the platform focus on the interconnection of the platform with the functional verification environment working directly in FPGA that allows automation and speed-up of checking the correctness of the system after the injection of faults.

3.2.2 Paper II

ČEKAN Ondřej, PODIVÍNSKÝ Jakub and KOTÁSEK Zdeněk. Software Fault Tolerance: the Evaluation by Functional Verification. In: *Proceedings of the 18th Euromicro Conference on Digital Systems Design.* Funchal: IEEE Computer Society, 2015, pp. 284-287. ISBN 978-1-4673-8035-5.

Author participation: 50% | Citations: 4 | Conference rank: B1 (Qualis)

Abstract

The aim of this paper is to present a new approach in evaluating Software Fault Tolerance (SFT) methodologies. It is the way on how to ensure fault tolerance without any additional hardware as is common in frequently used Triple Modular Redundancy (TMR). As our research is focused on electromechanical systems which are commonly driven by processors or Multi Processors Systems on Chip (MPSoC) we decided to use the soft-core processor running on Field Programmable Gate Array (FPGA) as our experimental platform. The new approach uses Functional Verification for automation of the evaluation process. The functional verification environment is one of the important parts of the presented evaluation platform architecture. Programs generation for a processor, where SFT is applied, is also important. Experiments with the programs generator and fault injection are presented and goals for future work are identified on that basis.

3.2.3 Paper III

PODIVÍNSKÝ Jakub, ČEKAN Ondřej, LOJDA Jakub and KOTÁSEK Zdeněk. Verification of Robot Controller for Evaluating Impacts of Faults in Electro-mechanical Systems. In: *Proceedings of the 19th Euromicro Conference on Digital Systems Design.* Limassol: IEEE Computer Society, 2016, pp. 487-494. ISBN 978-1-5090-2816-0.

Author participation: 35% | Citations: 0 | Conference rank: B1 (Qualis)

Abstract

Functional verification is a modern approach to verifying that a digital system complies with its specification. The verification environment for functional verification of robot controller which searches path for the robot through a maze is presented in this paper. This verification environment is designed according to UVM (Universal Verification Methodology) principles. As an interesting feature of the verification environment we see the use of

a mechanical part (robot in a maze) simulation. The article describes the use of the verification environment for evaluating impacts of faults in electro-mechanical systems. It will serve as a tool for automating the fault tolerance evaluation of electro-mechanical systems and together with the fault injector will form the basis of the verification platform in the future. The experimental results gained from the verification process are also presented in the paper.

3.2.4 Paper IV

ČEKAN Ondřej and KOTÁSEK Zdeněk. A Probabilistic Context-Free Grammar Based Random Test Program Generation. In: *Proceedings of 20th Euromicro Conference on Digital System Design*. Wien: Technical University Wien, 2017, pp. 356-359. ISBN 978-1-5386-2145-5.

Author participation: 95% | Citations: 2 | Conference rank: B1 (Qualis)

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.

3.2.5 Paper V

ČEKAN Ondřej, PODIVÍNSKÝ Jakub and KOTÁSEK Zdeněk. Program Generation Through a Probabilistic Constrained Grammar. In: *Proceedings - 21st Euromicro Conference on Digital System Design, DSD 2018*. Praha: IEEE Computer Society, 2018, pp. 214-220. ISBN 978-1-5386-7376-8.

Author participation: 90% | Citations: 1 | Conference rank: B1 (Qualis)

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 experimental 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 it is very suitable for an assembly code generation and universal use in the area of test stimuli.

3.2.6 Paper VI

ČEKAN Ondřej, PÁNEK Richard and KOTÁSEK Zdeněk. Input and Output Generation for the Verification of ALU: a Use Case. In: *Proceedings of 2018 IEEE East-West Design and Test Symposium, EWDTS 2018*. Kazan: IEEE Computer Society, 2018, pp. 331-336. ISBN 978-1-5386-5710-2.

Author participation: 85% | Citations: 0 | Conference rank: unknown

Abstract

The paper presents the approach to universal stimuli generation for an arithmetic-logic unit (ALU). It is not focused only on input data generation, but it is possible to generate also expected output in one stimulus. The process of generation is based on a probabilistic constrained grammar which is designed to universally describe stimuli for various circuits. This grammar is processed by our framework. The experiment in functional verification, which shows the quality of generated stimuli, is also presented.

3.2.7 Author's contributions to selected papers

Since the papers, on which this dissertation is based, were created in collaboration with other members of the Diagnostics research group led by supervisor docent Kotásek, the author's contributions to selected papers are explicitly listed in this chapter.

- **Paper I** - design of universal stimuli architecture, specific input structures and definition of constraints for assembler generation, functional verification of processor. Details were presented in subchapter 3.1.1 and 3.1.2.
- **Paper II** - development of approach for software implemented fault tolerance based on stimuli which contain triplicated instructions. More was discussed in subchapter 3.1.3.
- **Paper III** - definition of input structures for maze generation, functional verification of robot controller searching path in generated mazes. Details were presented in subchapter 3.1.4.
- **Paper IV** - development of general principles of generation based on newly defined formal grammar - probabilistic constrained grammar. General principles were defined in subchapter 3.1.5.
- **Paper V** - definition of the probabilistic constrained grammar for assembler code generation and its functional verification. Definitions were part of subchapter 3.1.6.
- **Paper VI** - definition of the probabilistic constrained grammar for arithmetic logic unit, generating both inputs and output. Details were presented in subchapter 3.1.6.

3.3 List of Other Publications

The author's other publications are listed for completeness in this subchapter, where they are sorted by year of publication.

2020

- LOJDA Jakub, PÁNEK Richard, PODIVÍNSKÝ Jakub, ČEKAN Ondřej, KRČMA Martin and KOTÁSEK Zdeněk. Analysis of Software-Implemented Fault Tolerance: Case Study on Smart Lock. In: *2020 IEEE East-West Design and Test Symposium, EWDTS 2020 - Proceedings*. Varna: Institute of Electrical and Electronics Engineers, 2020, pp. 24-28. ISBN 978-1-7281-9899-6.

  Author participation: 8%

- LOJDA Jakub, PODIVÍNSKÝ Jakub, ČEKAN Ondřej, PÁNEK Richard, KRČMA Martin and KOTÁSEK Zdeněk. Automatic Design of Reliable Systems Based on the Multiple-choice Knapsack Problem. In: *Proceedings - 2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2020*. Novi Sad: Institute of Electrical and Electronics Engineers, 2020, pp. 1-4. ISBN 978-1-7281-9938-2.

  Author participation: 10% | Conference rank: B3 (Qualis)

- PODIVÍNSKÝ Jakub, LOJDA Jakub, PÁNEK Richard, ČEKAN Ondřej, KRČMA Martin and KOTÁSEK Zdeněk. *Evaluation Platform For Testing Fault Tolerance: Testing Reliability of Smart Electronic Locks. In: 2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*. San José: IEEE Circuits and Systems Society, 2020, pp. 1-4. ISBN 978-1-7281-3427-7.

  Author participation: 10% | Conference rank: B5 (Qualis)

- LOJDA Jakub, PÁNEK Richard, PODIVÍNSKÝ Jakub, ČEKAN Ondřej, KRČMA Martin and KOTÁSEK Zdeněk. Hardening of Smart Electronic Lock Software against Random and Deliberate Faults. In: *Proceedings - Euromicro Conference on Digital System Design, DSD 2020*. Kranj: Institute of Electrical and Electronics Engineers, 2020, pp. 680-683. ISBN 978-1-7281-9535-3.

  Author participation: 12% | Conference rank: B1 (Qualis)

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, KRČMA Martin, BURGET Radek, HRUŠKA Tomáš and KOTÁSEK Zdeněk. Iterative Algorithm for Multidimensional Pareto Frontiers Intersection Determination. In: *2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS)*. San José: IEEE Circuits and Systems Society, 2020, pp. 1-4. ISBN 978-1-7281-3427-7.

  Author participation: 20% | Conference rank: B5 (Qualis)

2019

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, KRČMA Martin, BURGET Radek, HRUŠKA Tomáš and KOTÁSEK Zdeněk. Multidimensional Pareto Frontiers Intersection Determination and Processor Optimization Case Study. In: *Proceedings of the 2019 22nd Euromicro Conference on Digital System Design*. Kalithea: Institute of Electrical and Electronics Engineers, 2019, pp. 597-600. ISBN 978-1-7281-2861-0.

Author participation: 20% | Conference rank: B1 (Qualis)

- ČEKAN Ondřej, PODIVÍNSKÝ Jakub, LOJDA Jakub, PÁNEK Richard, KRČMA Martin and KOTÁSEK Zdeněk. Testing Reliability of Smart Electronic Locks: Analysis and the First Steps Towards. In: *Proceedings of the 2019 22nd Euromicro Conference on Digital System Design*. Kalithea: Institute of Electrical and Electronics Engineers, 2019, pp. 506-513. ISBN 978-1-7281-2861-0.

Author participation: 19% | Conference rank: B1 (Qualis)

2018

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, KRČMA Martin, BURGET Radek, HRUŠKA Tomáš and KOTÁSEK Zdeněk. A Framework for Optimizing a Processor to Selected Application. In: *Proceedings of IEEE East-West Design & Test Symposium*. Kazan: IEEE Computer Society, 2018, pp. 564-574. ISBN 978-1-5386-5710-2.

Author participation: 20% | Citations: 1

- PODIVÍNSKÝ Jakub, LOJDA Jakub, ČEKAN Ondřej and KOTÁSEK Zdeněk. Evaluation Platform for Testing Fault Tolerance Properties: Soft-core Processor-based Experimental Robot Controller. In: *Proceedings of the 2018 21st Euromicro Conference on Digital System Design*. Praha: IEEE Computer Society, 2018, pp. 229-236. ISBN 978-1-5386-7376-8.

Author participation: 24% | Citations: 1 | Conference rank: B1 (Qualis)

- LOJDA Jakub, PODIVÍNSKÝ Jakub, ČEKAN Ondřej, PÁNEK Richard and KOTÁSEK Zdeněk. FT-EST Framework: Reliability Estimation for the Purposes of Fault-Tolerant Systems Design Automation. In: *Proceedings of the 2018 21st Euromicro Conference on Digital System Design*. Praha: IEEE Computer Society, 2018, pp. 244-251. ISBN 978-1-5386-7376-8.

Author participation: 20% | Conference rank: B1 (Qualis)

2017

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, LOJDA Jakub, ŠIMKOVÁ Marcela, KRČMA Martin and KOTÁSEK Zdeněk. Functional Verification Based Platform for Evaluating Fault Tolerance Properties. *Microprocessors and Microsystems*, vol. 52, no. 5, 2017, pp. 145-159. ISSN 0141-9331.

Author participation: 18% | Citations: 4 | Impact factor: 1.045 (Q3)

- PODIVÍNSKÝ Jakub, LOJDA Jakub, ČEKAN Ondřej, PÁNEK Richard and KOTÁSEK Zdeněk. Reliability Analysis and Improvement of FPGA-based Robot Controller. In: *Proceedings of the 2017 20th Euromicro Conference on Digital System Design*. Wien: IEEE Computer Society, 2017, pp. 337-344. ISBN 978-1-5386-2145-5.

Author participation: 13% | Citations: 2 | Conference rank: B1 (Qualis)

2016

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, LOJDA Jakub and KOTÁSEK Zdeněk. Functional Verification as a Tool for Monitoring Impact of Faults in SRAM-based FPGAs. In: *Proceedings of the 2016 International Conference on Field Programmable Technology*. Xi'an: IEEE Computer Society, 2016, pp. 293-294. ISBN 978-1-5090-5602-6.

  Author participation: 22% | Citations: 1

- ČEKAN Ondřej. Generování testovacích stimulů. In: *Počítačové architektury a diagnostika PAD 2016*. Bořetice - Kraví Hora: Faculty of Information Technology BUT, 2016, pp. 97-100. ISBN 978-80-214-5376-0.

  Author participation: 100%

- ČEKAN Ondřej, PODIVÍNSKÝ Jakub and KOTÁSEK Zdeněk. Random Stimuli Generation Based on a Stochastic Context-Free Grammar. In: *Proceedings of the 2016 International Conference on Field Programmable Technology*. Xi'an: IEEE Computer Society, 2016, pp. 295-296. ISBN 978-1-5090-5602-6.

  Author participation: 90%

2015

- PODIVÍNSKÝ Jakub, ŠIMKOVÁ Marcela, ČEKAN Ondřej and KOTÁSEK Zdeněk. FPGA Prototyping and Accelerated Verification of ASIPs. In: *IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits and Systems*. Belgrade: IEEE Computer Society, 2015, pp. 145-148. ISBN 978-1-4799-6780-3.

  Author participation: 12% | Citations: 5 | Conference rank: B3 (Qualis)

- ČEKAN Ondřej. Principy generování verifikačních stimulů. In: *Počítačové architektury a diagnostika PAD 2015*. Zlín: Faculty of Applied Informatics, Tomas Bata University in Zlín, 2015, pp. 13-18. ISBN 978-80-7454-522-1.

  Author participation: 100%

- ČEKAN Ondřej, ŠIMKOVÁ Marcela and KOTÁSEK Zdeněk. Universal Pseudorandom Generation of Assembler Codes for Processors. In: *Proceedings of The Third Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale*. Grenoble: COST, European Cooperation in Science and Technology, 2015, pp. 70-73.

  Author participation: 70%

2014

- ČEKAN Ondřej, ŠIMKOVÁ Marcela and KOTÁSEK Zdeněk. Solving of Constraint Satisfaction Problem. In: *Proceedings of the 20th Conference STUDENT EEICT 2014. Volume 3*. Brno: Faculty of Information Technology BUT, 2014, pp. 291-295. ISBN 978-80-214-4924-4.

Author participation: 65%

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, ŠIMKOVÁ Marcela and KOTÁSEK Zdeněk. The Evaluation Platform for Testing Fault-Tolerance Methodologies in Electro-mechanical Applications. In: *17th Euromicro Conference on Digital Systems Design*. Verona: IEEE Computer Society, 2014, pp. 312-319. ISBN 978-1-4799-5793-4.

Author participation: 30% | Conference rank: B1 (Qualis)

- ČEKAN Ondřej. Universal Generation of Test Vectors for Functional Verification. In: *Počítačové architektury a diagnostika 2014*. Liberec: Liberec University of Technology, 2014, pp. 44-49. ISBN 978-80-7494-027-9.

Author participation: 100%

3.4 Participation in Research Projects and Grants

- FIT-S-20-6309 — *Design, Optimization and Evaluation of Application Specific Computer Systems*, Brno University of Technology, team member.
- 8A18014, Proposal ID 783119-2 — *SECREDAS - Product Security for Cross Domain Reliable Dependable Automated Systems*, ECSEL Joint Undertaking, team member.
- FIT-S-17-3994 — *Advanced parallel and embedded computer systems*, Brno University of Technology, team member.
- LQ1602 — *IT4Innovations excellence in science*, MŠMT CZ, team member.
- 7H14002, 621439 — *Algorithms, Design Methods, and Many-Core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing*, Artemis Joint Undertaking, team member.
- FIT-S-14-2297 — *Architecture of parallel and embedded computer systems*, Brno University of Technology, team member.
- LD12036 — *Methodologies for Fault Tolerant Systems Design Development, Implementation and Verification*, MŠMT CZ, team member.
- ED1.1.00/02.0070 — *The IT4Innovations Centre of Excellence*, MŠMT CZ, team member.

Chapter 4

Discussion and Conclusions

This chapter summarizes the results achieved in this dissertation and suggests a possible further direction for the research in the field of stimuli generation.

The research presented in this thesis focused on the design of general principles in the field of generating stimuli for various systems. Stimuli represent the input data of a system that determines its behavior. A significant advantage is the use of these principles in the field of functional verification. How the stimulus generator can be involved in the whole process of verifying the correct behavior was presented in the theoretical part of the thesis, where the three most used techniques of functional verification were introduced. Functional verification is one of the verification techniques that, based on simulation, verifies the correct behavior of the system by monitoring its inputs and outputs. After a detailed study of already available stimuli generators, criteria were determined that completely differentiate the architecture proposed by the author from the methods available so far. There are four criteria - parameterizability, speed, randomness, versatility. Parameterizability is ensured by separate input definitions with the possibility of change during verification. Significant speed is given by obtaining a valid input without evaluating the semantics of the partial components of the stimulus. Valid input is obtained by solving a constraint satisfaction problem. Randomness and universality are determined by selecting a suitable general description using formal grammar with the possibility of balancing probabilities. The acquired knowledge was summarized in Chapter 2, which serves as a starting point for the following research.

The research started with the design of a universal stimuli generation architecture, which aims to generally define the widest possible set of stimuli. The basis of this architecture are two input descriptions that define the desired stimuli format and the constraints imposed on it. Thanks to this, it is possible to obtain both a valid stimulus and change the constraints during generation, which is especially suitable for obtaining higher coverage in functional verification. The architecture designed in this way was already used in all other researches of the author.

Input descriptions were further defined for the proposed generation method. These input descriptions were created to verify the processor and generate assembler programs. The instruction set of the available RISC and VLIW processors was encoded into the input description along with the constraints that ensured a valid sequence of instructions, preservation of the prescribed latencies, memory accesses, jumps and labels. In total, 18 types of constraints were proposed for these processors, which ensured the construction of a valid program. The programs obtained in this way were verified by a functional verification. The acquired knowledge was applied in the evaluation of software fault tolerance, where the

individual instructions of the program were secured by both time and space redundancy. Thanks to this, it was possible to correct single and partly also multiple faults in the program memory.

The research continued by verifying the control unit of a robot searching for a path in a maze. One of the input data, that the robot uses, is a bitmap image of the maze in which the robot searches for a path. Input descriptions were designed for such a maze in order to verify the behavior of the robot in the event of a fault in various situations (mazes). When designing the input descriptions, it was necessary to take into account the dimensions of the maze, the start and goal position of the robot, the continuity of the corridors and the existence of a path from any selected point to another. Based on this information, a set of mazes was generated, on which the behavior of the robot in functional verification was verified. It was also necessary to extend the specific input descriptions with additional constraints.

The following part of the research focused on the generalization of previously defined input descriptions using the bottom-up method. For this purpose, a formal grammar was used, which is able to generate a certain language. The already existing probabilistic context-free grammar was used and extended by constraints, which created a completely new grammar with a higher expressive power than context-free grammar. In this dissertation, the new grammar is called probabilistic constrained grammar. This grammar allows the preservation of the generation context necessary to ensure the construction of a valid stimulus. All previously described system stimuli have been encoded into this new grammar thanks to a one general description. Descriptions of stimuli for other systems have also been added (ALU and stepper motor), which underline the generality of the whole proposal. The generation process has also been accelerated, improved and validated in functional verification. In addition to the versatility of the whole solution compared to conventional approaches, the generation in this way is faster and achieves comparable or better results of coverage in functional verification.

Based on the findings, the principles of creating stimuli for verified systems were described, and moreover, are also applicable in other similar systems. These principles are an important output of this dissertation.

4.1 Contributions

Based on the objectives of the dissertation, this subchapter summarizes the main benefits presented in this thesis:

1. **Architecture of universal stimuli generation**
   A new architecture specially designed to generate stimuli was introduced. The architecture extends the classic functional verification scheme by two additional blocks that define the description of the required stimuli. This makes it possible to control the generation during verification process and achieve better coverage results. This architecture was introduced in Paper I.

2. **Generation using formal grammar**
   A new formal grammar has been proposed based on solving the constraint satisfaction problem. Formal grammar allows for a general description while constraints provide the necessary expressive power and behavior to generate valid stimuli. The newly designed grammar is called the Probabilistic Constrained Grammar. With the help of

constraints, it is possible to change the validity of production rules during generation and ensure the use of only those rules that are valid for a given generated sequence. The new grammar was introduced in Paper V.

3. **Input descriptions for different systems**
   A set of input descriptions defining stimuli for different digital systems was proposed. First, a great attention was paid to processors for which specific constraints were created. Experimentation was also with software fault tolerance. (Paper II). The robot controller required a maze for its operation (Paper III). After generalization using formal grammar (Paper IV), descriptions were also created for other systems - ALU (Paper VI) and stepper motor.

4.2 Future Work

Based on the findings, possible future research directions in the field of stimuli generation were identified:

- One of the challenges for defining more complex structures is to generate a hardware description language (HDL). The main idea is to design a certain grammar that can be easily transformed into a corresponding finite state machine. The finite state machine can then be transformed directly into an HDL using an existing tool (e.g. Kiss2-to-VHDL [1]). Such a description can be directly analyzed and synthesized.

- There are a number of languages for describing digital systems (VHDL, Verilog [62], Java HDL [8], Active-HDL [26], and more). If it was possible to use this description as input and automatically generate at least a part of the production rules of formal grammar or constraints, it would greatly speed up the whole process of creating descriptions of stimuli and thus reduce the time needed to develop the system.

- The current stimuli generation always ends at the specified number of lines. A significant advantage and optimization would be the introduction of a request-response method, which would return one line of the generated stimulus based on the request (in the case of a processor, it would be one instruction). The finality of the generation would thus be determined by the system for which the given stimulus is intended.

Bibliography

[1] Abdel Hamid, A., Zaki, M. and Tahar, S. A Tool Converting Finite State Machine to VHDL. *Canadian Conference on Electrical and Computer Engineering.* june 2004, vol. 4, p. 1907 – 1910 Vol.4.

[2] Acarnley, P. *Stepping Motors: A Guide to Theory and Practice.* Institution of Engineering and Technology, 2002. Control, Robotics and Sensors. Available at: https://books.google.cz/books?id=mwDCORmy6u0C. ISBN 9780852964170.

[3] Accellera. *Universal Verification Methodology (UVM) 1.2 User's Guide [online].* October 2015 [cit. 2020-05-11]. Available at: https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf.

[4] Amiri, A. M., Khouas, A. and Boukadoum, M. Pseudorandom Stimuli Generation for Testing Time-to-Digital Converters on an FPGA. *IEEE Transactions on Instrumentation and Measurement.* 2009, vol. 58, no. 7, p. 2209–2215.

[5] Ashenden, P. *The VHDL Cookbook [online].* Adelaide: Department of Computer Science, University of Adelaide, 1990 [cit. 2015-01-02]. Available at: http://www.ics.uci.edu/~alexv/154/VHDL-Cookbook.pdf.

[6] Ayub, M. A., Kalpoma, K. A., Proma, H. T., Kabir, S. M. and Chowdhury, R. I. H. Exhaustive study of essential constraint satisfaction problem techniques based on N-Queens problem. In: *2017 20th International Conference of Computer and Information Technology (ICCIT).* 2017, p. 1–6.

[7] Belkin, V. and Sharshunov, S. 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, p. 73–74.

[8] Bellows, P. and Hutchings, B. JHDL-an HDL for reconfigurable systems. In: *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251).* 1998, p. 175–184.

[9] Brailsford, S. C., Potts, C. N. and Smith, B. M. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research.* 1999, vol. 119, no. 3, p. 557 – 581. ISSN 0377-2217.

[10] Ceschia, M., Violante, M., Reorda, M., Paccagnella, A., Bernardi, P. et al. Identification and classification of single-event upsets in the configuration memory of SRAM-based FPGAs. *Nuclear Science, IEEE Transactions on.* 2003, vol. 50, no. 6, p. 2088–2094. ISSN 0018-9499.

[11] Chomsky, N. Three models for the description of language. *IRE Transactions on Information Theory*. 1956, vol. 2, no. 3, p. 113–124.

[12] Codasip. *Codix Cores [online]*. 2014 [cit. 2015-01-06]. Available at: https://www.codasip.com/products/cores/.

[13] Corno, F., Reorda, M., Squillero, G. and Violante, M. 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, p. 195–198. ISBN 0-7695-0909-6.

[14] Corno, F., Sanchez, E., Reorda, M. and Squillero, G. Automatic test program generation: a case study. *IEEE Design and Test of Computers*. march 2004, vol. 21, no. 2, p. 102–109. ISSN 0740-7475.

[15] Dandamudi, S. P. *Guide to RISC Processors*. Springer-Verlag New York, 2005. ISBN 978-0-387-21017-9.

[16] Elakkiya, C., Murty, N. S., Babu, C. and Jalan, G. Functional Coverage - Driven UVM Based JTAG Verification. In: *2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*. 2017, p. 1–7.

[17] Evans, A., Silburt, A., Vrckovnik, G., Brown, T., Dufresne, M. et al. Functional Verification of Large ASICs. In: *Proceedings of the 35th Annual Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 1998, p. 650–655. DAC '98. Available at: https://doi.org/10.1145/277044.277210. ISBN 0897919645.

[18] Fine, S. and Ziv, A. Coverage directed test generation for functional verification using Bayesian networks. In: *Proceedings of the 40th Design Automation Conference (DAC 2003)*. June 2003, p. 286–291.

[19] Foster, P. R. Verification and validation of a diffraction program. In: *IEE Colloquium on Application and Validation of Design Tools for Antennas*. 1993, p. 4/1–4/5.

[20] George, M. and Ait Mohamed, O. Performance analysis of constraint solvers for coverage directed test generation. In: *The 23rd International Conference on Microelectronics (ICM 2011)*. Elsevier Science, 2011, p. 1–5.

[21] Giegerich, R. Introduction to Stochastic Context Free Grammars. In: Gorodkin, J. and Ruzzo, W. L., ed. *RNA Sequence, Structure, and Function: Computational and Bioinformatic Methods*. Totowa, NJ: Humana Press, 2014, p. 85–106. Available at: https://doi.org/10.1007/978-1-62703-709-9_5. ISBN 978-1-62703-709-9.

[22] Gohel, D. Pure SV Verification Environment Methodology for ASIC Verification. may 2014, vol. 5, no. 5, p. 770–775. ISSN 2229-5518.

[23] Guglielmo, G. D., Guglielmo, L. D., Fummi, F. and Pravadelli, G. Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs. *Journal of Electronic Testing*. 2011, vol. 27, no. 2, p. 137–162. ISSN 1573-0727.

[24] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T. et al. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. Washington, DC, USA: IEEE Computer Society, 2001, p. 3–14. WWC '01. Available at: http://dx.doi.org/10.1109/WWC.2001.15. ISBN 0-7803-7315-4.

[25] Hany, A., Ismail, A., Kamal, A. and Badran, M. Approach for a unified functional verification flow. In: *2013 Saudi International Electronics, Communications and Photonics Conference*. 2013, p. 1–6.

[26] Haskell, R. E. and Hanna, D. M. *Digital Design Using Digilent FPGA Boards Verilog/Active-HDL Edition*. 2nd ed. LBE Books, 2012. ISBN 9780980133776.

[27] Hatnik, U. and Altmann, S. Using ModelSim, Matlab/Simulink and NS for Simulation of Distributed Systems. In: *International Conference on Parallel Computing in Electrical Engineering (PARELEC 2004)*. September 2004, p. 114–119.

[28] Hatnik, U. and Altmann, S. Using ModelSim, Matlab/Simulink and NS for Simulation of Distributed Systems. In: *Parallel Computing in Electrical Engineering, 2004. PARELEC 2004. International Conference on*. September 2004, p. 114–119.

[29] Hlavička, J. *Číslicové systémy odolné proti poruchám*. Praha: ČVUT, 1992. 330 p. ISBN 8001008525, 9788001008522.

[30] Hopcroft, J. E., Rajeev, M. and D., U. J. *Introduction to Automata Theory, Languages, and Computation*. Pearson, 2006. ISBN 9780321462251.

[31] Hudec, J. 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*. June 2011, p. 527–532. ISSN 1330-1012.

[32] IEEE. Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*. 2018, p. 1–1315.

[33] Ju, A. L. and Spasojevic, M. Smart Jewelry: The Future of Mobile User Interfaces. In: *Proceedings of the 2015 Workshop on Future Mobile User Interfaces*. New York, NY, USA: Association for Computing Machinery, 2015, p. 13–15. FutureMobileUI '15. Available at: https://doi.org/10.1145/2754633.2754637. ISBN 9781450335041.

[34] Jäger, G. and Rogers, J. Formal language theory: refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences*. 2012, vol. 367, no. 1598, p. 1956–1970. Available at: https://royalsocietypublishing.org/doi/abs/10.1098/rstb.2012.0077.

[35] Kajan, M. *Verifikace číslicových obvodu [online]*. UPSY FIT VUT v Brně, 2012 [cit. 2015-01-02]. Available at: http://www.fit.vutbr.cz/units/UITS/grants/index.php?file=%2Fproj%2F556%2Fpcs_verifikace_extended.pdf&id=556.

[36] Karlapalem, S. and Venugopal, S. Scalable, Constrained Random Software Driven Verification. In: *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. 2016, p. 71–76.

[37] Koren, I. and Krishna, C. M. *Fault-Tolerant Systems*. San Francisco: Morgan Kaufmann Publishers Inc., 2007. 378 p. ISBN 0120885255, 9780080492681.

[38] Kotthoff, L. Constraint Solvers: An Empirical Evaluation of Design Decisions. *ArXiv e-prints*. january 2010, p. 1–23.

[39] Kropf, T. *Introduction to Formal Hardware Verification*. New York, USA: Springer, 1999. 299 p. ISBN 9783540654452.

[40] Kumar, R. *Theory Of Automata Languages Computation*. McGraw-Hill Education (India) Pvt Limited, 2010. ISBN 9780070702042.

[41] Kumar, V. Algorithms for Constraint Satisfaction Problems: A Survey. *AI MAGAZINE*. 1992, vol. 13, no. 1, p. 32–44.

[42] MathWork®. *MATLAB and Simulink* [https://www.mathworks.com/]. 2018. Accessed: 2019-03-20.

[43] Meduna, A. *Formal Languages and Computation*. Taylor & Francis Informa plc, 2014. 315 p. Taylor and Francis. Available at: https://www.fit.vut.cz/research/publication/10524. ISBN 978-1-4665-1345-7.

[44] Mehta, A. B. *ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies*. 1st ed. Springer Publishing Company, Incorporated, 2017. ISBN 3319594176.

[45] Mehta, A. B. SystemVerilog Functional Coverage (SFC). In: *ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies*. Cham: Springer International Publishing, 2018, p. 129–148. Available at: https://doi.org/10.1007/978-3-319-59418-7_7. ISBN 978-3-319-59418-7.

[46] Meyer, A. *Principles of Functional Verification*. Amsterdam: Elsevier Science, 2003. 216 p. ISBN 978-0-7506-7617-5.

[47] Miller, F. P., Vandome, A. F. and McBrewster, J. *Cyclic Redundancy Check: Computation of CRC, Mathematics of CRC, Error Detection and Correction, Cyclic Code, List of Hash Functions, Parity Bit, Information ... Cksum, Adler- 32, Fletcher's Checksum*. Alpha Press, 2009. ISBN 6130219741.

[48] Mizuno, F. and Nishihara, S. Local minimum structures of graph-coloring problems for stochastic constraint satisfaction algorithms. In: *Proceedings 12th IEEE Internationals Conference on Tools with Artificial Intelligence. ICTAI 2000*. 2000, p. 366–369.

[49] Monfroy, E., Crawford, B. and Soto, R. Automatic Triggering of Constraint Propagation. In: Murgante, B., Misra, S., Carlini, M., Torre, C. M., Nguyen, H.-Q. et al., ed. *Computational Science and Its Applications – ICCSA 2013*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 452–461. ISBN 978-3-642-39640-3.

[50] Oliveira, R., Jagirdar, A. and Chakraborty, T. J. A TMR Scheme for SEU Mitigation in Scan Flip-Flops. In: *ISQED '07: Proceedings of the 8th International Symposium on Quality Electronic Design*. Washington, DC, USA: IEEE Computer Society, 2007, p. 905–910. ISBN 0-7695-2795-7.

[51] Padmanabhuni, S. Extended analysis of intelligent backtracking algorithms for the maximal constraint satisfaction problem. In: *Engineering Solutions for the Next Millennium. 1999 IEEE Canadian Conference on Electrical and Computer Engineering (Cat. No.99TH8411)*. 1999, p. 1710–1715 vol.3.

[52] Patterson, D. A. Reduced Instruction Set Computers. *Commun. ACM*. New York: ACM. january 1985, vol. 28, no. 1, p. 8–21. ISSN 0001-0782.

[53] Reid, S. The Art of Software Testing, Second edition. Glenford J. Myers. Revised and updated by Tom Badgett and Todd M. Thomas, with Corey Sandler. John Wiley and Sons, New Jersey, U.S.A., 2004. ISBN: 0-471-46912-2, pp 234. *Software Testing, Verification and Reliability*. 2005, vol. 15, no. 2, p. 136–137. Available at: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.322.

[54] Rodrigues, C. A case study for Formal Verification of a timing co-processor. In: *2009 10th Latin American Test Workshop*. 2009, p. 1–6.

[55] Ronacher, A. *Jinja¶ [online]*. The Pallets Projects, 2015 [cit. 2020-05-11]. Available at: https://jinja.palletsprojects.com/.

[56] Roy, S. and Ramesh, S. Functional verification of system on chips - practices, issues and challenges. In: *Proceedings of ASP-DAC 2002*. 2002, p. 11–13.

[57] Russel, J. and Cohn, R. *Arithmetic Logic Unit*. Book on Demand, 2013. Available at: https://books.google.cz/books?id=l6SBMAEACAAJ. ISBN 9785510817393.

[58] Schubert, K. POWER7 — Verification challenge of a multi-core processor. In: *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*. 2009, p. 809–812.

[59] Storey, N. R. *Safety critical computer systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[60] Surhone, L., Tennoe, M. and Henssonow, S. *Triple Modular Redundancy*. Betascript Publishing, 2010. Available at: https://books.google.cz/books?id=NjumcQAACAAJ. ISBN 9786132988683.

[61] Tasiran, S. and Keutzer, K. Coverage metrics for functional validation of hardware designs. *Design Test of Computers, IEEE*. june 2001, vol. 18, no. 4, p. 36–45. ISSN 0740-7475.

[62] Thomas, D. and Moorby, P. *The Verilog Hardware Description Language*. 5th ed. Springer US, 2002. ISBN 9780387849300.

[63] Yuan, J., Pixley, C. and Aziz, A. *Constraint-based verification*. New York: Springer, 2006. I-XII, 1-253 p. ISBN 978-0-387-25947-5.

Appendices

Appendix A

Publications cited by other authors

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, ŠIMKOVÁ Marcela and KOTÁSEK Zdeněk. The Evaluation Platform for Testing Fault-Tolerance Methodologies in Electro-mechanical Applications. *Microprocessors and Microsystems*, vol. 39, no. 8, 2015, pp. 1215-1230. ISSN 0141-9331.

  - MCWILLIAM, R.; KHAN, S.; FARNSWORTH, M.; et al.: Zero-maintenance of Electronic Systems: Perspectives, Challenges, and Opportunities. *Microelectronics Reliability*, volume 85, 2018: pp. 122–139

  - HAO, Z.; ZHANG, M.: Cultivation of Mechanical Application Talents Based on FPGA and Machine Learning. *Microprocessors and Microsystems*, IN PRESS, 2020, ISSN 0141-9331.

- ČEKAN Ondřej, PODIVÍNSKÝ Jakub and KOTÁSEK Zdeněk. Software Fault Tolerance: the Evaluation by Functional Verification. In: *Proceedings of the 18th Euromicro Conference on Digital Systems Design.* Funchal: IEEE Computer Society, 2015, pp. 284-287. ISBN 978-1-4673-8035-5.

  - SYED RIFFAT, A.: Reliability Testing for Advanced Networks. In: *Next Generation and Advanced Network Reliability Analysis.* Springer, Cham, 2019. pp. 277-304. ISBN 978-3-030-01646-3.

  - ZHANG, T.; WANG, J.: A Spatial-Temporal Model for Software Fault Tolerance in Safety-Critical Applications. *In: 2017 IEEE International Conference on SoftwareQuality, Reliability and Security Companion (QRS-C)*, IEEE, 2017, pp. 575–576.

  - ZHANG, T.; WANG, X.: High-Reliable Testing for FPGA Software in Space Utilization Engineering. *In: 2017 International Conference on Dependable Systems and Their Applications (DSA)*, IEEE, pp. 86–91.

  - CAI, B.; LIU, Y.; LIU, Z.; et al..:Bayesian Networks for Reliability Engineering. Springer, 2020.

- PODIVÍNSKÝ Jakub, ŠIMKOVÁ Marcela, ČEKAN Ondřej and KOTÁSEK Zdeněk. FPGA Prototyping and Accelerated Verification of ASIPs. In: *IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits and Systems*. Belgrade: IEEE Computer Society, 2015, pp. 145-148. ISBN 978-1-4799-6780-3.

  - JORDANS, R.; JÓŹWIAK, L.; CORPORAAL, H.; et al.: Automatic Instruction-set Architecture Synthesis for VLIW Processor Cores in the ASAM Project. *Microprocessors and Microsystems*, volume 51, 2017: pp. 114–133.

  - CABA, J.; CARDOSO, J. M.; RINCÓN, F.; et al.: Rapid Prototyping and Verification of Hardware Modules Generated Using HLS. *In: International Symposium on Applied Reconfigurable Computing*, Springer, 2018, pp. 446–458.

  - CABA, J.; RINCÓN, F.; DONDO, J.; et al.: Testing Framework For in-hardware Verification of the Hardware Modules Generated Using HLS. *In: 2018 28th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, IEEE,2018, pp. 103–110.

  - CABA, J.; RINCÓN, F.; DONDO, J.; et al.: Testing Framework for on-board Verification of HLS Modules Using Grey-box Technique and FPGA Overlays. *Integration*, volume 68, 2019: pp. 129–138.

  - CABA, J.; RINCÓN, F.; BARABA, J.; et al.: FPGA-Based Solution for On-Board Verification of Hardware Modules Using HLS. *Electronics*, volume 9, no. 12, 2020.

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, LOJDA Jakub and KOTÁSEK Zdeněk. Functional Verification as a Tool for Monitoring Impact of Faults in SRAM-based FPGAs. In: *Proceedings of the 2016 International Conference on Field Programmable Technology*. Xi'an: IEEE Computer Society, 2016, pp. 293-294. ISBN 978-1-5090-5602-6.

  - PENG, X.; et al.: Function Verification of SRAM Controller Based on UVM. In: *2019 IEEE 13th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. Xiamen, China, 2019, pp. 1-5, doi: 10.1109/ICASID.2019.8925105.

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, LOJDA Jakub, ŠIMKOVÁ Marcela, KRČMA Martin and KOTÁSEK Zdeněk. Functional Verification Based Platform for Evaluating Fault Tolerance Properties. *Microprocessors and Microsystems*, volume 52, no. 5, 2017, pp. 145-159. ISSN 0141-9331.

  - LIU, X.; YOUAN, G.; QIAO, S.: Accelerating Functional Verification for Digital Circuit with FPGA Hard Processor System. *Journal of Electronics Information Technology*, volume 41, no. 5, 2019.

  - QAMAR, S.; BUTT, W. H.; ANWAR, M. W.; et al.: A Comprehensive Investigation of Universal Verification Methodology (UVM) Standard for Design

Verification. *In: Proceedings of the 2020 9th International Conference on Software and Computer Applications*, 2020, pp. 339–343.

  - SABAMONIRI, S.; SOURI, A.: A Weighted Resource Discovery Approach in Grid Computing. *International Journal of Pervasive Computing and Communications*, 2019.

  - SOURI, A.; RAHMANI, A. M.; NAVIMIPOUR, N. J.; et al.: A Symbolic Model Checking Approach in Formal Verification of Distributed Systems. *Human-centric Computing and Information Sciences*, volume 9, no. 1, 2019: pp. 4.

- ČEKAN Ondřej and KOTÁSEK Zdeněk. A Probabilistic Context-Free Grammar Based Random Test Program Generation. In: *Proceedings of 20th Euromicro Conference on Digital System Design*. Wien: Technical University Wien, 2017, pp. 356-359. ISBN 978-1-5386-2145-5.

  - PAVESE, ESTEBAN, et al. Inputs from Hell: Generating Uncommon Inputs from Common Samples. *arXiv preprint* arXiv:1812.07525, 2018.

  - SOREMEKUN, EZEKIEL, et al. Inputs from Hell Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering*, 2020. doi: 10.1109/TSE.2020.3013716.

- PODIVÍNSKÝ Jakub, LOJDA Jakub, ČEKAN Ondřej, PÁNEK Richard and KOTÁSEK Zdeněk. Reliability Analysis and Improvement of FPGA-based Robot Controller. In: *Proceedings of the 2017 20th Euromicro Conference on Digital System Design*. Wien: IEEE Computer Society, 2017, pp. 337-344. ISBN 978-1-5386-2145-5.

  - RUIZ-ROSERO, J.; RAMIREZ-GONZALEZ, G.; KHANNA, R.: Field Programmable Gate Array Applications — A Scientometric Review. *Computation*, volume 7, no. 4, 2019: p. 63.

  - WANG, L.; WANG, X.; JIA, P.; et al.: Reliability, Safety and Time - Domain Sensitivity Analysis of Double 2-out-of-2 Redundancy System Based on Markov Process and Multiple Beta Factor Model. *In: 2018 3rd International Conference on System Reliability and Safety (ICSRS)*, IEEE, 2018, pp. 153–161.

- ČEKAN Ondřej, PODIVÍNSKÝ Jakub and KOTÁSEK Zdeněk. Program Generation Through a Probabilistic Constrained Grammar. In: *Proceedings - 21st Euromicro Conference on Digital System Design, DSD 2018*. Praha: IEEE Computer Society, 2018, pp. 214-220. ISBN 978-1-5386-7376-8.

  - FITZPATRICK, G.: Mind the gap: Modelling the human in human-centric computing. *In: 2018 IEEE Symposium on Visual Languages and Human-Centric*

Computing (VL/HCC). IEEE Computer Society, 2018. p. 3-3.

- PODIVÍNSKÝ Jakub, LOJDA Jakub, ČEKAN Ondřej and KOTÁSEK Zdeněk. Evaluation Platform for Testing Fault Tolerance Properties: Soft-core Processor-based Experimental Robot Controller. In: *Proceedings of the 2018 21st Euromicro Conference on Digital System Design*. Praha: IEEE Computer Society, 2018, pp. 229-236. ISBN 978-1-5386-7376-8.

  – RUIZ-ROSERO, J.; RAMIREZ-GONZALEZ, G.; KHANNA, R.: Field Programmable Gate Array Applications — A Scientometric Review. *Computation*, volume 7, no. 4, 2019: p. 63.

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, KRČMA Martin, BURGET Radek, HRUŠKA Tomáš and KOTÁSEK Zdeněk. A Framework for Optimizing a Processor to Selected Application. In: *Proceedings of IEEE East-West Design & Test Symposium*. Kazan: IEEE Computer Society, 2018, pp. 564-574. ISBN 978-1-5386-5710-2.

  – Mazurek P.: BOSON - Application-Specific Instruction Set Processor (ASIP) for Educational Purposes. In: *2020 16th International Conference on Control, Automation, Robotics and Vision (ICARCV)*. Shenzhen, China, 2020, pp. 1323-1328, doi: 10.1109/ICARCV50220.2020.9305396.

Related Papers

Paper I

The Evaluation Platform for Testing Fault-Tolerance Methodologies in Electro-mechanical Applications

PODIVÍNSKÝ Jakub, ČEKAN Ondřej, ŠIMKOVÁ Marcela, KOTÁSEK Zdeněk

Microprocessors and Microsystems, vol. 39, no. 8, 2015, pp. 1215-1230. ISSN 0141-9331.

Microprocessors and Microsystems 39 (2015) 1215–1230

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

The evaluation platform for testing fault-tolerance methodologies in electro-mechanical applications

Jakub Podivinsky *, Ondrej Cekan, Marcela Simkova, Zdenek Kotasek

Brno University of Technology, Faculty of Information Technology, Bozetechova 2, 612 66 Brno, Czech Republic

ARTICLE INFO

Article history:
Available online 30 May 2015

Keywords:
Fault-tolerance
Electro-mechanical systems
Fault injection
Single event upset
Functional verification

ABSTRACT

The aim of this paper is to present a new platform for estimating the fault-tolerance quality of electro-mechanical (EM) systems based on FPGAs. We demonstrate one working example of such an EM system that was evaluated using our platform: the mechanical robot and its electronic controller in an FPGA. Different building blocks of the electronic robot controller allow us to model different effects of faults on the whole mission of the robot (searching a path in a maze). In the experiments, the mechanical robot is simulated in a simulation environment, where the effects of faults artificially injected into its controller can be seen. In this way, it is possible to differentiate between the fault that causes the failure of the system and the fault that only decreases its performance. Further extensions of the platform focus on the interconnection of the platform with the functional verification environment working directly in FPGA that allows for the automation and speed-up for checking the correctness of the system after the injection of faults.

1. Introduction

In several areas, such as aerospace and space applications or automotive safety–critical applications, fault-tolerant electro-mechanical (EM) systems are highly desirable. In these systems, the mechanical part is controlled by its electronic controller. Currently, the trend is to add even more electronics into EM systems. For example, in aerospace, extending of the electronic part results in a lower weight that helps to reduce operating costs [1] [2]. The situation is similar in other sectors, such as automotive sg. [3].

It is obvious that the fault-tolerance methodologies are targeted mainly to the electronic components because they perform the actual computation. However, as the electronics can be realized on different hardware platforms (ASICs, FPGAs, etc.), specific fault-tolerance techniques dedicated for these platforms must be developed.

The previous activities of the team at our department specialized on fault tolerant systems design are described in [4]. In that paper, the fault tolerant methodology for the SRAM based FPGA based on the use of Partial Dynamic Reconfiguration and the Generic Partial Dynamic Reconfiguration Controller inside the FPGA were presented.

The goal of our present research is to develop a platform for the verification of EM systems resilience against faults which occur in an electronic component controlling the system, the component is designed as fault tolerant. Besides from this main activity, the use of functional verification for the automated evaluation of fault impacts is described. The goals are available in details in Section 3.

Our research is targeted to *Field Programmable Gate Arrays* (FPGAs) [5] as they present many advantages from the industrial point of view. They can compute many problems hundreds times faster than modern microprocessors while their reconfigurability allows the same flexibility as microprocessors. FPGAs can be either programmed before their use or reconfigured during program runtime of circuit. Partial dynamic reconfiguration can be also used when programming is performed only on a part of the circuit, while the rest of the circuit is working. The programmability of FPGA differs from Application Specific Integrated Circuit (ASIC) to which the required function was configured in its production cycle. FPGAs are becoming increasingly popular and are used in many applications, mainly due to their programmability, ease of design, flexibility, decreasing power consumption and price. The robot manipulator presented in [6], or the FPGA-based robot arm controller presented in [7], can serve as an example. Moreover, the National Instruments company presents their power train controls which also use FPGAs on their web [8]. They are used mainly in the applications where it is necessary to produce small series and design of ASIC and solution with microprocessor is inappropriate.

* Corresponding author.
E-mail addresses: ipodivinsky@fit.vutbr.cz (J. Podivinsky), icekan@fit.vutbr.cz (O. Cekan), isimkova@fit.vutbr.cz (M. Simkova), kotasek@fit.vutbr.cz (Z. Kotasek).

http://dx.doi.org/10.1016/j.micpro.2015.05.011
0141-9331/

FPGAs can be used advantageously for prototyping complex custom devices. Programmability can also be used to change the behavior of the circuit by a customer which allows to correct errors in design or to add new features to circuit already in use.

FPGAs are composed of *Configurable Logic Blocks* (CLBs) that are interconnected by a programmable interconnection net. Every CLB consists of *Look-Up Table* (LUT) that realizes the logic function, a multiplexer and a flip-flop. The structure of FPGA and CLB is shown in Fig. 1. The configuration of CLBs and of the interconnection net is stored in the SRAM memory. Except CLBs, FPGA contains advanced circuits and other elements, such as *Block Memory* (BRAM), fast multipliers or *Digital Signal Processors* (DSPs). *Input/Output Blocks* (IOBs) can be used as the FPGA communication interface.

The problem from the reliability point of view is that FPGAs are quite sensitive to faults caused by charged particles [9]. These particles can induce an inversion of a bit in the configuration SRAM memory of an FPGA (or directly to its internal flip-flops) and this may lead to a change in its behavior. Affecting SRAM or directly the flip-flops can be seen as equivalent in possible consequences. This event is called the *Single Event Upset* (SEU). That is the reason why so many fault-tolerance methodologies inclined to FPGAs have been developed and new ones are under investigation which is mentioned in Section 2.

We decided to use FPGAs in our research mainly because of their speed, re-configurability and because we aim to evaluate various fault-tolerant methodologies dedicated to FPGAs. Despite our exemplary system is not so complex as typical FPGA applications are, it serves for evaluating these methodologies connected to the verification environment very well. All our previous research in the area of fault tolerant systems design was oriented to FPGAs and all our tools were developed for this platform. Therefore, the system presented in this paper has been physically also realized on FPGA mainly for our research purposes and not because it cannot be realized on different platforms as well (for instance, on an ASIC or on a microprocessor).

The paper is organized as follows. The basic concepts connected to the FPGA reliability and verification of hardware systems are summarized in Section 2. The goals of our research and the interconnection scheme of the platform for estimating the quality of EM systems can be found in Section 3. The architecture of our experimental design, the robot controller, is provided in Section 4. A detailed description of the fault injection process that is used for artificial injection of faults into the robot controller is described in Section 5.1. Results of the experiments with the robot controller are available in Section 5.2. The future work that includes using *functional verification* for automated evaluation of impacts of faults and the stimuli generation process is presented in Sections 6 and 7. Section 8 presents another use case – the processor, the reliability of which will be checked in our future work. Finally, the paper is concluded in Section 9.

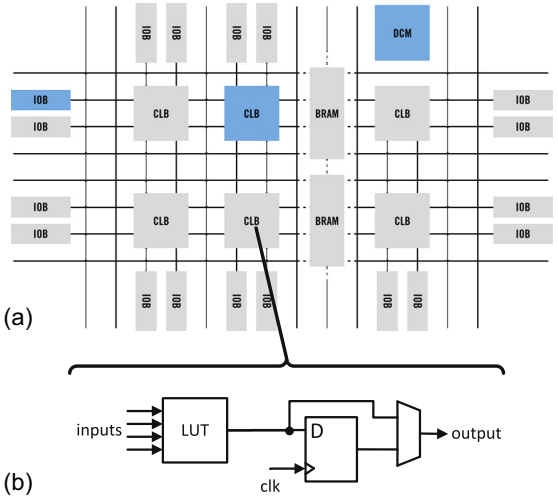


Fig. 1. Structure of (a) *Field Programmable Gate Array* (FPGA) and (b) *Configurable Logic Blocks* (CLB).

The research was supported by the following European projects: EU COST Action IC1103 - MEDIAN – "Manufacturable and Dependable Multicore Architectures at Nanoscale" and project IT4Innovations Centre of Excellence (ED1.1.00/02.0070).

2. Related work

Our presented research is unique in a combination of fault-tolerance methodologies and functional verification for improving the reliability of digital systems. For a better understanding, the reader should be familiar with the basic concepts and trends in these two areas. The basic overview is outlined in this section.

2.1. Fault-tolerance methodologies for FPGAs-based systems

Fault-tolerance (FT) is an important feature for many systems, especially for those that aim to be highly reliable. A fault-tolerant system is also able to operate correctly in the presence of faults (SEUs, transient faults, etc.). There are several basic FT architectures that use hardware redundancy such as n-modular redundancy or duplex systems [10]. A special type of n-modular redundancy is *Triple Modular Redundancy* (TMR) which is able to mask a single fault in the system. TMR uses three identical copies of a functional unit (FU) and the unit called Voter. If there is a fault in one FU, Voter chooses the output value using a majority function applied on the primary outputs of the FUs. The TMR architecture is shown in Fig. 2a.

The duplex architecture also provides fault security and is used as the core of many advanced FT architectures. The duplex system can be seen in Fig. 2b. It uses two identical copies of a FU and a comparator (XOR). The output signal *error* informs us about a fault occurrence in the system.

The other type of redundancy, which can be used for hardening against faults, is time redundancy [11]. Time redundancy is based on the repetitive result calculation using the same components but at different time intervals. The obtained results are then compared together. If there are differences, a fault is detected. The scheme of time redundancy is shown in Fig. 3.

The presented hardware redundancy is able to mask a fault occurrence in the FT system. However, the fault localization is needed in order to repair the faulty modules. For these purposes, techniques called *Concurrent Error Detection* (CED) were developed. These techniques encapsulate on-line checkers, self-checking units or parity checkers. A combination of the duplex system with CED that is based on time redundancy is presented in [12]. The duplex system is able to detect a fault occurrence. If a fault is detected, recomputation in the next time slot is able to locate the faulty module. In comparison to the presented TMR architecture, this approach saves some resources. The use of time redundancy as CED leads to less power consumption because the result is recomputed only if a fault is detected. Moreover, this technique reduces the number of input and output pins of the combinational logic.

An important feature of FPGAs, which can be utilized for reliability purposes after a fault (we consider SEUs) is detected, is called *Partial Dynamic Reconfiguration* (PDR) [13]. PDR allows for modifying

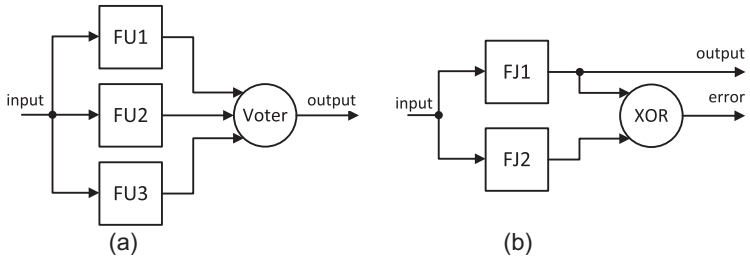


Fig. 2. Architectures with hardware redundancy: (a) TMR and (b) the duplex architecture.

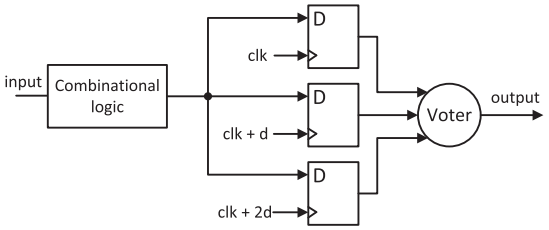


Fig. 3. Time redundancy basic scheme for combinational logic.

or reloading a specified part of the FPGA configuration memory while the rest of the FPGA is working correctly. Prepared parts of configuration memory can be stored in an external memory and read when they are needed. For example, a part of the FPGA can operate as a multiplier after the initialization, but the reconfiguration process can change the function of this part to an adder.

PDR can also reconfigure the affected part of the FPGA (a faulty module) and restore the electronic system into the correct operation without interrupting the other parts of the system. The recovery of a faulty module in TMR by using reconfiguration is illustrated in Fig. 4. If one of three FUs of TMR is faulty, TMR still provides correct output values and the faulty module (FU3) can be repaired by PDR without stopping the FPGA operation. Moreover, if another module (FU1) is faulty, then TMR produces incorrect output values. Due to the reconfiguration, these faulty modules can be repaired and TMR is able to produce the correct output.

Sensitivity to faults (especially SEUs) and the possibility of reconfiguration are the main reasons why so many fault-tolerance methodologies inclined to FPGAs have been developed and new ones are under investigation [14,15]. Our plan is to test the usability and quality of these methodologies (and also their new alternatives) while hardening FPGA-based controllers of mechanical systems against faults.

2.2. Testing fault-tolerant systems implemented on FPGAs

The weak point of FPGAs from the reliability point of view is their configuration memory. The functionality of an FPGA chip is defined by the sequence of configuration bits (called *bitstream*) which is loaded into the configuration memory. In our case, a specific part of bitstream determines the functionality of the robot controller. However, even the smallest change in the configuration memory can lead to a different functionality. When a charged particle strikes a memory cell, the resulting effect is the inversion of the stored value (SEU) [16].

During the testing of the resilience of systems against faults, waiting for their natural appearance is not feasible. A typical reason is the *Mean Time Between Failures* (MTBF) parameter that can be in the order of years. Therefore, some special techniques were developed in order to artificially accelerate the fault occurrence.

The accurate simulation method for the emulation of the effects of SEUs in the configuration memory of FPGAs is presented in [17]. This approach combines simulation and topological analysis of the design mapped on the FPGA. An analytical algorithm is presented which is able to accurately identify the electrical effects induced into the resources of the circuit affected by a SEU. This simulator avoids designers to use an expensive FPGA board, but there is a problem that the design is not evaluated on a real target platform (FPGA).

An FPGA-based fault injection tool, which is presented in [18], supports several synthetizable fault models of digital systems and is implemented using VHDL. The authors present a real time fault injection tool with good controllability and observability. However, the fault injection requires an addition of some extra gates and wires to the original design and thus modifying the original VHDL. There are several types of faults that can be generated. For example, the model of injecting SEU can be seen in Fig. 5. There are additional signals Bit and FIS which are connected to the Fault injection component (implemented on the same FPGA). A weak point of this approach is the difference between the Device Under Test (DUT) and the device which will be manufactured.

In [19,20], techniques which are based on the fault injection into a real FPGA board without changing of the original design were presented. These techniques are based on PDR which allows us to read the configuration bitstream, inverse bits and write the affected bitstream back to the FPGA. The prototype of the evaluation board for the fault injection purposes was presented in [19]. There are two FPGAs, the first one is used as the DUT and the second one is used as the fault injection controller. In [20] the authors

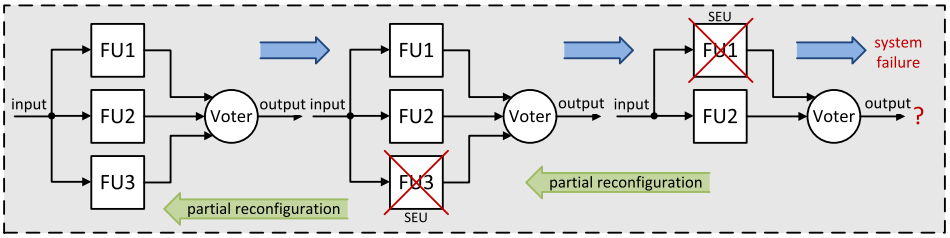


Fig. 4. Recovery of the faulty module in TMR by PDR.

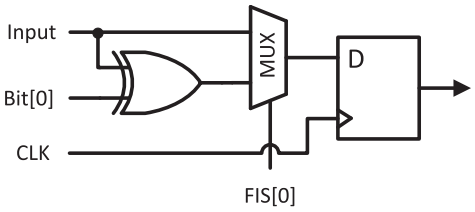


Fig. 5. The synthetizable SEU model [18].

present FLIPPER. This fault injection platform is composed of two boards with FPGAs – the main board and the DUT board. The fault injection is controlled by the main board which is driven by the software application running on a PC. It is able to use various types of FPGAs as the DUT board, but only if there are enough input/output pins on the main board. The authors in [21] focus on the speed of the fault impact evaluation, where the fault injection is fully controlled by a part of the design on the FPGA. Communication with a PC is used only for the initial configuration of the fault injection process.

Our previous research also covered the artificial injection of faults and we have developed an external SEU injector that is described in more detail in [22]. This injector is based on the SEU generation outside of the FPGA (in PC), so it is not targeted to a specific FPGA board (testing was performed on the ML506 card with the Virtex 5 FPGA technology). The original and the modified bitstream is transported through the JTAG interface and the subsequent dynamic reconfiguration of the FPGA. The process of the SEU generation is divided into four steps: (1) specifying location of the fault injection, (2) reading the related part of the configuration bitstream, (3) the SEU generation = inversion of the specified bit of the bitstream and (4) applying the bitstream using PDR without stopping the FPGA. Our fault injector is implemented in TCL in two basic layers, the structure of which is shown in Fig. 6. The first layer (Bitstream Generation Layer) is responsible for

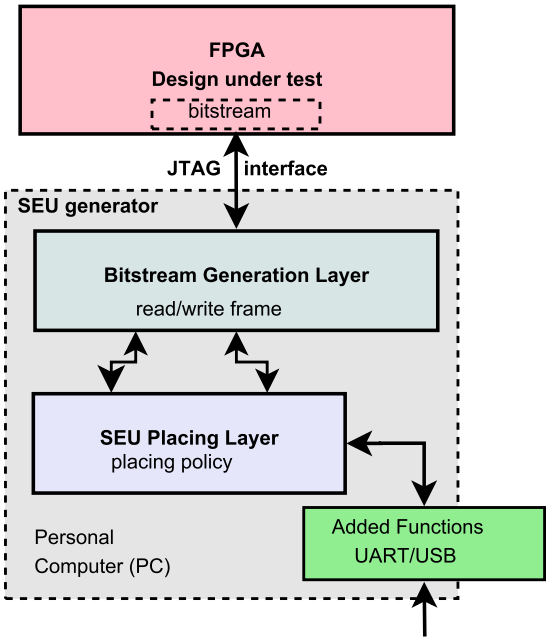


Fig. 6. An external SEU injector structure [22].

communication with the FPGA through the standard JTAG interface and uses ChipScope libraries. The SEU injection layer is responsible for the read and write bitstream according to the specified fault location. The last block (Added Functions) makes it possible to drive the SEU generation by external sources, such as an external program or the UART interface.

2.3. Verification of hardware systems

Verification is the process of checking whether a model of the hardware system satisfies a given correctness specification. Verification is an important phase in the development of hardware systems because, before the system is taped-out to the silicon, it is desirable to detect all design and functional errors or the misinterpretations of the specification as early as possible. Moreover, as the hardware complexity has grown rapidly in the last decade, verification is even more important, but very time-consuming too.

Verification methods provide ideally yes/no answers, thus informing about correctness or incorrectness of the system. There are two basic types of verification – formal verification and functional verification. Both aim at verifying the system functionality according to the specification.

Formal verification [23] verifies the system by using mathematical methods in order to formally describe the system and on the basis of logical formulas to prove the correctness of the system. Functional verification [24] verifies the system by monitoring the inputs and outputs in the simulation environment (usually RTL simulators are used). For a thorough verification of the system, a huge number of pseudo-random stimuli is needed in order to cover all key properties of the system.

There is little space to thoroughly compare both of the above mentioned verification approaches and to mention their pros and cons. But in general, functional verification is easier to apply for hardware engineers as they are familiar with simulation tools and this approach does not require a deep knowledge of formal specifications. Moreover, standard languages, methodologies and libraries were defined for functional verification. The most commonly known are the SystemVerilog IEEE language standard, Universal Verification Methodology (UVM) and the open-source UVM library (with all the basic components of verification environments). On the other hand, formal verification is more precise. In our work, we use functional verification. The main concepts of this approach are mentioned in the following paragraphs.

At this point, it is important to mention the difference between verification and testing the system against injected faults. Verification is mostly the part of the pre-silicon development and aims of design errors. Testing against faults is usually done after verification and usually with real hardware representation of the system (e.g. FPGA). The reason is that when we inject faults into the system and the system does not behave correctly, we must be sure that the failure is caused by the injected fault and not by some design error still present in the system. Therefore, we will distinguish design under verification (DUV) in the verification phase and DUT in the testing phase.

In functional verification, the DUV outputs are compared to the outputs of the reference model (sometimes also referred to as the golden model) that is typically implemented by a verification engineer or a designer who did not implement the DUV. This is very important because the interpretation of the specification that is done by two (or more) different people is actually compared. If a discrepancy between the two models is detected, an error in the system, or at least any suspicious behavior, can be discovered. The basic principle of functional verification is demonstrated in Fig. 7. An important prerequisite for functional verification is also a good generator of stimuli for verifying all interesting scenarios depicted by the specification.

There are three basic methods on how stimuli are produced and applied on the inputs of DUV.

The first method [25] (see Fig. 8) uses a random stimuli generator, which generates a set of stimuli without any control. Uncovered key functions are covered by directed tests, which have to be created manually by a verification engineer based on the coverage analysis. The coverage analysis is an output from the simulation environment (an RTL simulator supporting functional verification) and contains information about the coverage of the key functions and lines of code of DUV too. The main disadvantage of this method is that it generates a large amount of invalid input tests.

The second method is called constraint random stimuli generation (CRSG) [26] (see Fig. 9). Since we are interested in certain scenarios when we are verifying the system, by using CRSG we can generate specific and always valid stimuli that satisfy predefined constraints and target these scenarios. To be more specific, these constraints represent inputs for the constraint solver. The constraint solver is a unit which solves defined constraints and generates valid stimuli. Some parts of the verified circuit may remain uncovered, hence additional constraints or directed tests have to be specified manually as in the previous method.

The last method is called coverage-directed stimuli generation (CDSG) [27], also called coverage-driven verification (see Fig. 10) and is characterized by some kind of automation. This method is based on CRSG and moreover, it uses data from the coverage analysis in order to direct the next round of input stimuli generation and to cover unverified areas of the system.

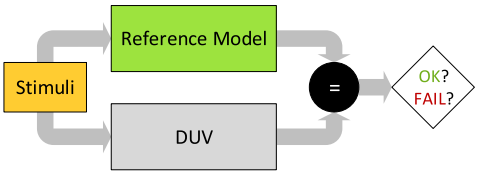


Fig. 7. The main principle of functional verification.

2.4. Constrained-random test generation

As was mentioned above in the previous subsection, functional verification works with constraints. A generator, which is based on solving constraints, is also known as the constraint solver. Its task is to search such an assignment of a value to each variable so that all imposed constraints are simultaneously satisfied. Solving constraint-random stimuli generation in functional verification is equivalent to solving the NP-hard problem called Constraint Satisfaction Problem (CSP).

Constraint Satisfaction Problem [28,29] is a general mathematical problem defined as a set of variables which can take values from a finite and discrete domain and a set of constraints. The constraint is defined on a subset of variables and determines values from the domain that a variable can take. The result is a solution of one or all evaluations of variables so that the constraints are satisfied. Among the typical examples of CSPs are N Queens problem, Map-Coloring problem, Car sequencing problem, Magic Square, Social Golfers, etc.

The implementation of constraint-random stimuli generator that effectively manages CSP and its parameters can be set or modified in runtime is highly desirable. Therefore, a part of our research is supposed to be targeted to this domain.

3. The goals of the research

We have identified two areas that we would like to focus on in our research of fault-tolerant FPGA-based systems: the first one is that methodologies are validated and demonstrated only on simple electronic circuits implemented in FPGAs. For instance, methodologies focusing on the memory in [30] are validated on simple memories without any additional logic around. In [31], the fault-tolerance technique is presented only on a two-input

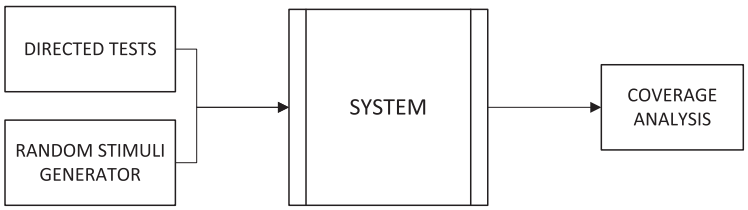


Fig. 8. The method with random stimuli generator.

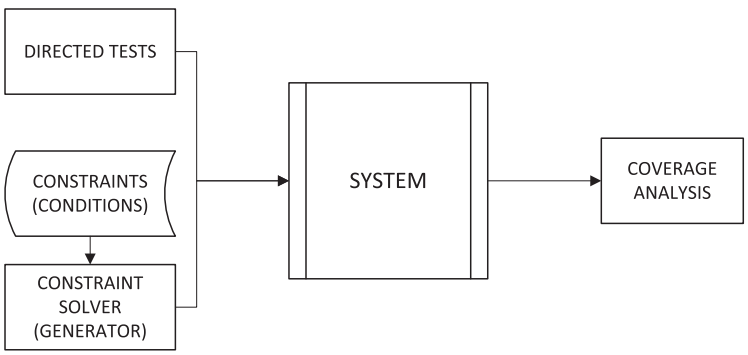


Fig. 9. The method with constrained-random stimuli generator.

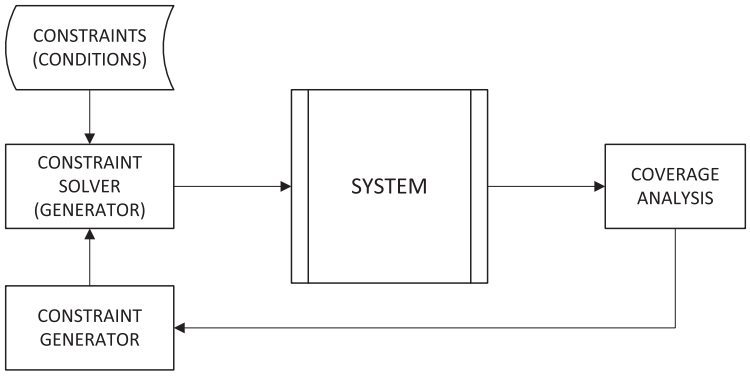


Fig. 10. The method coverage directed stimuli generation.

multiplexer, one simple adder and one counter. Other methodology dedicated to harden finite state machines [32] is only applied on a simple finite state machine. Of course, for demonstration purposes, such circuits are satisfactory. However, in real systems different types of blocks must be protected against faults at the same time and must communicate with each other. Therefore, a general evaluation platform for testing, analysis and comparison of alone-working or cooperating fault-tolerance methodologies is needed.

As for the second area of research and the main contribution of our work, we feel that it must be possible to check the reactions of the mechanical part of the system if the functionality of its electronic controller is corrupted by faults. It is either done through simulation or by a physical realization.

According to the identified problems we have formulated our goals in the following way:

1. *Developing an evaluation platform based on the FPGA technology for checking the resilience of EM applications against faults.*
2. *Developing and verifying a new methodology for increasing fault-tolerance qualities of EM applications using the proposed platform.*

Under the term EM system a mechanical device and its electronic controller implemented in an FPGA is understood. In our experiments, these components are represented by a robot device and its controller, which drives the movement of a robot in a maze.

At this point, we also wanted to target the issue of complexity. The electronic part, the robot controller, is designed as a complex system with specific components that will allow testing and validating individual or cooperating fault-tolerance methodologies based on the FPGA.

As for the first goal of our research, we have already implemented the evaluation platform that consists of three basic parts:

- the Virtex5 FPGA board, where the robot controller is situated after the synthesis and the place and route process;
- the simulation environment Player/Stage [33] for checking responses of the mechanical device to instructions from the robot controller (see Fig. 11);
- the external fault injector (PC) which inserts faults into the robot controller [22].

The second goal of our research is covered by the development of a methodology on how to incrementally harden EM systems against faults. We expect to clearly identify the situations when the reconfigurable hardware correctly covers its functions (and the robot works properly), but also the situations when the mechanical functions are corrupted and the robot collapses.

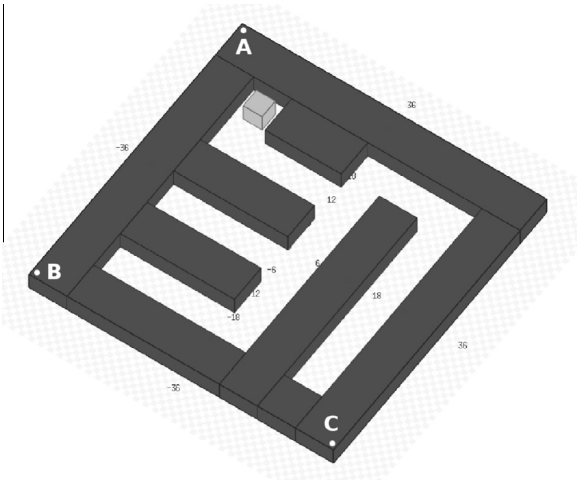


Fig. 11. The robot in a maze in player/stage simulation environment.

Fig. 12 shows the overall interconnection of the PC and the FPGA board in our platform. It should be noted that there are two devices called FITkit [34] in both directions, from the PC to the FPGA and vice versa. FITkit is a hardware platform that was developed for student projects at the Faculty of Information Technology, Brno University of Technology. In our platform, FITkits represent a communication layer and serve as a debugging point for communication between the PC and the FPGA board. The SEU injector runs on the PC and is connected through the JTAG interface directly to the main FPGA board where the robot controller is situated. Via the connection between the SEU injector and the simulation environment (as shown in Fig. 12), we are able to control the SEU injection process into the robot controller for every mission and to see the effects of faults directly in simulation.

In our opinion, it is important to find a relation between the level of functional corruption of the electronic controller and the corruption of the mechanical functionality in the EM systems (i.e. between the robot controller and the simulated mechanical robot). Therefore, it must be possible to introduce various levels of external faults into the controller and check whether the mechanical function: (a) was not corrupted, (b) was partially corrupted, or (c) was completely corrupted.

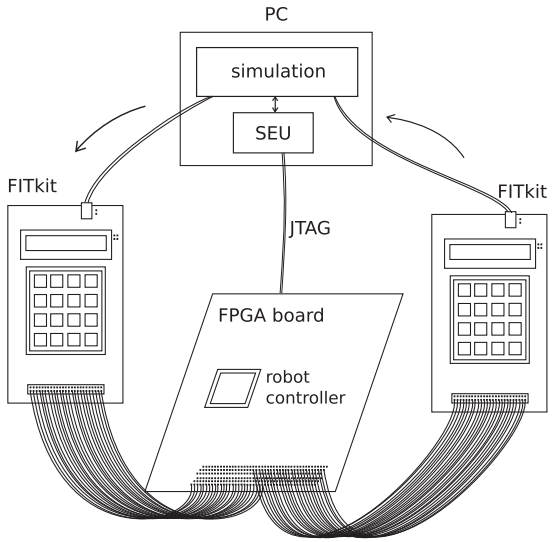


Fig. 12. The platform for testing fault-tolerance methodologies.

4. The robot controller – structure and principles

In Fig. 13, the block diagram of the implemented robot controller is outlined. The control unit is connected to the PC (where the simulation environment is located) via the Interface Block. Through this block, data from the simulation is received (information about barriers, distances from control points, target positions) and in the opposite direction, instructions about the movement of the robot are sent (direction and speed).

The robot controller is composed of various blocks, their function is described in [35]. Only the main characteristics of every component are summarized here. The central block of the robot controller is a bus through which the communication between each block is accomplished. Each of the component, without the Engine Control Module is connected to the bus. The Position Evaluation Unit acquires its distance from the control points, which are located in the fixed positions in the maze. From these, the position of the robot in the maze is calculated and provided to other units as coordinates x and y. The Barrier Detection Unit (BDU) uses four sensors; each located on one side of the robot (cubical robot) and provides information about the distance to the surrounding barriers. The output is a four-bit vector that represents the four-neighborhood of the robot and informs us about barriers in this area. Map updating is provided by the Map Unit (MU) and is based on information about the position of the robot obtained from the Position Evaluation Unit and information about the occurrence of barriers in a four-neighborhood provided by the Barrier Detection Unit. The Map Memory Unit (MMU) stores information about the up-to-date map. The memory is realized by the block memory (BRAM) available in the FPGA. The most important block that manages the activity of other blocks in the robot controller is the Path Finding Unit (PFU). It implements the simple iteration algorithm for finding a path through the maze according to the information about the current and the desired target position. The mechanical parts of the robot are driven by the setting of the speed in the required direction of the movement by the Engine Control Module (ECM).

The robot controller is designed as a complex system with specific components that will allow for testing and validating various types of fault-tolerant methodologies focused on FPGAs:

- *Combinational circuits*
  Combinational circuits are the basic types of digital circuits and their output is dependent just on the current input. In the robot controller, the Barrier Detection Unit represents a pure combinational circuit.

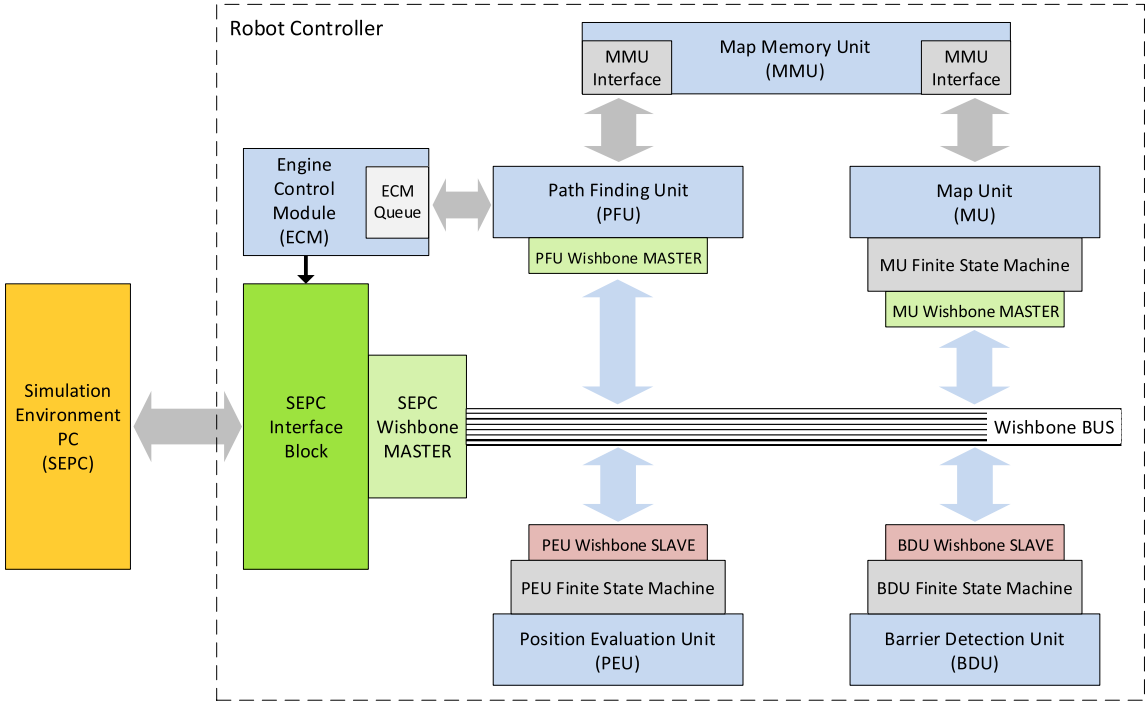


Fig. 13. The block diagram of the robot controller.

- *Sequential circuits*
  The output of the sequential circuit, unlike the combinational circuit, is not only dependent on the current input but also on the actual state. These circuits also contain memory for storing a state. Sequential circuits can be explicitly controlled by the finite state machine. Sequential circuits without an explicit control are represented by the Map Unit and the Position Evaluation Unit in the robot controller.
- *Finite state machines*
  Finite state machines also represent sequential circuits, their computational process is modeled by states and transitions between them. In the robot controller, the Path Finding Unit and the Engine Control Module, together with units that provide the bus communication, are implemented as finite state machines.
- *Buses*
  The bus is a central element of our controller. We decided to use a freely available *Wishbone bus* [36] that is configured as a shared bus. It means that the communication on the bus can be driven only by one master device and the other units must wait for releasing the bus. All function blocks are connected to the bus via their wrapper.
- *Memories*
  In the robot controller, we can find two occurrences of different types of memory. The first, the Map Memory Unit, is realized as the Block Memory (BRAM) which is available on the FPGA. The second memory is a queue in the Engine Control Module that stores a continuously calculated path to the destination.

5. Experiments with the robot controller

5.1. Evaluation of reliability by fault injection

In order to simulate the effects of faults in the FPGA, it could be done by a direct change of the configuration bitstream which is loaded into the configuration memory. For this purpose we implemented a fault injector [22] which allows us to prepare the bitstream for our FPGA and also to modify single or multiple bits of the bitstream in order to simulate single and multiple faults. As a consequence, the design placed in the FPGA (determined by the configuration data) is similarly influenced by a real fault which strikes the hardware architecture of the FPGA in a real environment.

For effective testing of fault effects on a system composed of several blocks, we need to determine the block in which the fault will be injected. In the case of injecting faults into the whole FPGA we are not sure which block is affected, or if the useful part of the bitstream is hit. The implemented injector is able to inject faults only to the specified bits of the configuration memory and a specification list of these bits is an input parameter.

The list of bits representing each component is obtained through several steps. First, we perform synthesis using Xilinx synthesis tools [37]. The result of synthesis is a netlist, which serves as an input for the next step. Next, we use the PlanAhead [38] tool for the layout of the components on the FPGA. Thanks to this, we know where each component is placed. The bitstream is generated in this step and the FPGA can be programmed. The knowledge about the component layout allows us to use the RapidSmith [39] tool for analysing the design. This tool is able to generate a list of the bitstream bits that correspond to the identified areas of the FPGA, while we know which components are in each area. The disadvantage is that this process only provides a list of bitstream bits that correspond to *Lookup Tables* (LUTs). Our goal in the future will be to find a method which allows us to also localize bits of the bitstream corresponding to the interconnection network.

5.2. Experimental results

The aim of the experiment is to identify which parts of the robot controller are vulnerable to faults. The flow of the experiment is displayed in Fig. 14. At first, the environment of the robot in simulation was initiated. We generated a maze together with the start and the end position for the mission of the robot. As the first scenario, we chose a small maze with 8×8 fields. The start position was in the upper left corner and the end position in the lower right corner. Subsequently, the robot controller is initiated. In particular, the bistream for the Virtex5 FPGA board is generated. When loaded, the robot starts to search a path to the end position. It moves quite slowly, one robot mission takes about one minute. At this point, the fault injection takes place. We generate randomly a LUT of every unit of the robot controller into which the fault will be injected. Thanks to the Rapidsmith, only corresponding bits of the bistream are inverted. We want to point out that only bits of the bitstream belonging to the robot controller design are targeted. Other bits of the bitstream belonging to the unused parts of the FPGA or to the interconnection network are not affected. Faults are injected one after another (MTBF = 2 s) until the robot starts to behave incorrectly or fails. We were monitoring (1) the number of faults that led to the malfunction of the robot and (2) how the behavior of the robot was changed.

The results of the experiments are shown in Table 1. In the first column, the list of components of the robot controller is provided. In the second column, the total number of bits of the bitstream that belong to the LUTs of corresponding components is shown. The following three columns represent the number of injected faults into particular components which caused the incorrect behavior of the robot. The first number is minimum, the second number is median and the last number is maximum of faults that led to failure. Injecting faults into all bits of the bitstream would be very time-consuming. Therefore, we utilized the statistic evaluation. Twenty experimental runs were performed for each component (320 experimental runs in total). The last column of the table contains the state of the robot that was evaluated as the wrong behavior. These states are described in more detail later in the text.

The statistical data from the measures are also demonstrated in Fig. 15. It is a quartile chart that for each component shows the minimum, the first quartile (25%), median, the second quartile (75%) and maximum of the measured number of injected faults that led to its failure. One interesting conclusion arises from the graph. The incorrect behavior did not appear immediately after the first injection of a fault. We can conclude that some bits of the bitstream, despite the fact that they are identified as related to the robot controller, are not used to store a useful information. This can be seen particularly in components PEU_FSM and PEU_WB, the numbers of injected faults were so high that they did not fit into the graph. There are several explanations for this (for example not all inputs of LUTs are employed or not all states of FSMs are visited during the computation). On the other hand, the components MU, MU_FS or MU_WB were corrupted by a

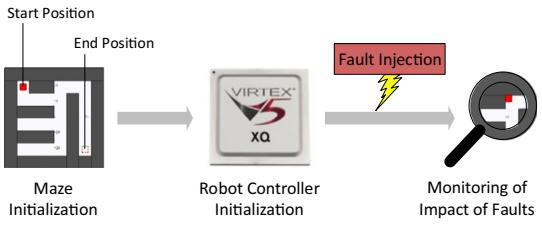


Fig. 14. The flow of one experiment.

Table 1
The experimental results.

Components	# Bits	Number of injected faults			Consequence
		Min	Med	Max	
PEU	21,632	2	6	12	Freezing
PEU_FSM	2112	>80	–	>80	–
PEU_WB	2112	41	–	>80	Freezing
BDU	320	2	6	21	Freezing
BDU_FSM	2752	3	6	34	Freezing
BDU_WB	2176	3	9	28	Freezing
SEPC_INF	1216	2	3	7	Freezing
SEPC_WB	9088	2	3	7	Freezing
ECM	25,664	1	2	7	Freezing
PFU	7488	3	6	12	Deadlock
PFU_WB	7424	2	3	9	Freezing
MU	11,840	1	2	3	Crashing
MU_FSM	1280	1	3	5	Freezing
MU_WB	7680	1	3	6	Freezing
MMU	3008	1	3	6	Freezing
WB_BUS	5056	1	3	6	Freezing

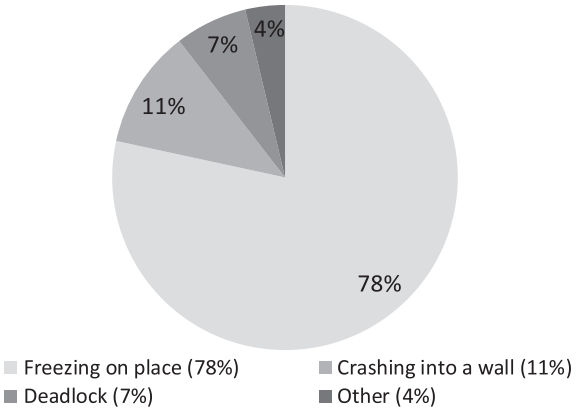


Fig. 16. The chart of typical consequences of injected faults on the mission of the robot.

relatively small number of faults. It means, that many bitstream bits store useful information. Therefore, we realized that some components contain more critical bits than others and thus they should be preferred while hardening against faults by some fault-tolerance methods.

The most common consequences of the injected faults are:

- *Freezing on place*
  Freezing on one spot means that the robot suddenly stopped after the fault injection and did not continue in its mission.
- *Deadlock*
  After the injection of a certain number of faults the robot began to walk around in a cycle.
- *Crashing into a wall*
  In some cases, the robot did not recognize the occurrence of walls in the maze and repeatedly crashed into the wall.
- *Other*
  In the experiments, we observed a small number of other interesting consequences of faults. An example might be freezing of the robot in one place, then a re-freezing or walking in a cycle. We also noted a wrong turn of the robot in the maze, which was followed by freezing.

The proportional representation of these consequences is displayed in Fig. 16. As can be deduced from the chart, the most common consequence of injected faults is *Freezing on place*. We can also conclude that the stopping of the robot is not so critical as for example, a collision with the wall. This conclusion can be very critical and useful for different kinds of EM systems.

6. The use of functional verification for automated evaluation of fault impacts

For extensive checking of the behavior of the robot or any other EM system placed into our evaluation platform, we need to examine various scenarios. After the application of proper stimuli, we can prove the correctness and accuracy of the behavior of the system with respect to the specification. The manual check of these stimuli is difficult as it requires full control from the user. The user is responsible for running the testbench, generating stimuli and also analysing the outputs of the system. All these activities are time-demanding and, therefore, it is not possible to examine the system thoroughly in a reasonable time. It is necessary to apply some kind of automation. An extended technique for automated checking of the correctness of the system is called functional verification and was described in Section 2.

In order to be able to inject faults into the FPGA while performing functional verification, we must carry out verification directly in the FPGA (not just in the simulation as usual). We can advantageously use and modify hardware accelerated verification that uses an FPGA as the acceleration board. An example of such an accelerator is the open-source framework HAVEN [40]. The extension of our evaluation platform with the support of functional verification is shown in Fig. 17. The DUT (in our case the robot controller) is placed on the FPGA. The outputs from the FPGA are compared to the outputs of the reference model and they also represent the inputs that are propagated to the simulation of the mechanical part. Thus, the output of the DUT stimulates the movement of the mechanical part of the robot in the simulated maze. The inputs

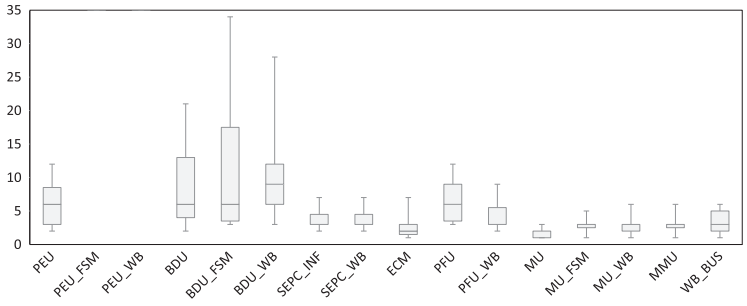


Fig. 15. The quartile graph of the results of experiments.

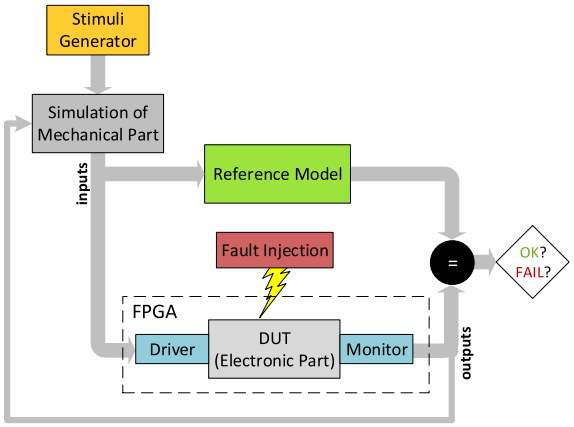


Fig. 17. The functional verification involvement in our platform with the fault injection.

for the FPGA and the reference model are data from the sensors of the mechanical part of the robot.

As the reference model, a second implementation of the control unit, for example in SystemVerilog, C, SystemC, or the same VHDL implementation that is used as the DUT but without injected faults, can be considered. The Fault Injector is a unit that differentiates the current proposal from the regular functional verification environment. By adding this feature we can verify that the fault-tolerance techniques used in the robot controller are working properly and the robot also behaves correctly in the presence of faults injected into its controller.

The verification process aiming at evaluating the quality of fault-tolerance methodologies in fault-tolerant EM systems that utilizes the fault injection is shown in Fig. 18. This process is divided into three main phases that are described below.

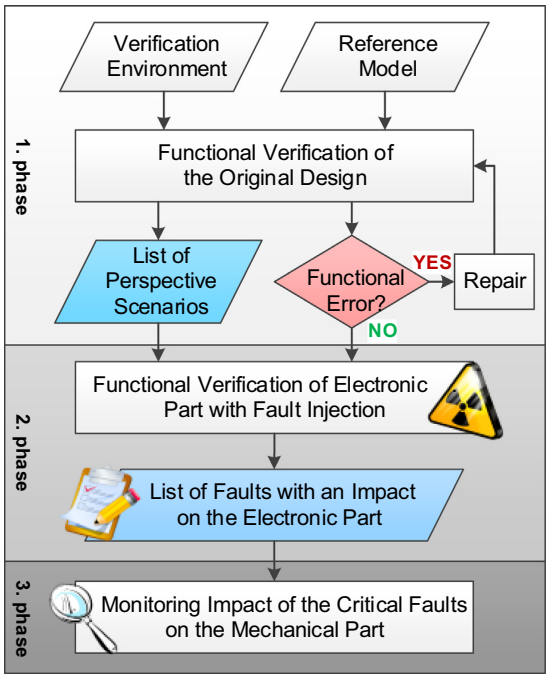


Fig. 18. The flow of phases in the FT systems verification.

At first, the verification environment and the reference model for the electronic control unit (the robot controller) must be created. In our case, we decided to use the reference model implemented in the C/C++ language. In the first phase, we use the regular simulation-based functional verification where the VHDL description of the electronic robot controller is used as the DUT. It is also important to connect the environment, where the mechanical parts of the robot are simulated, to the verification environment. For clarification, there are two simulation environments: functional verification is running in the RTL simulation environment and the mechanical robot is simulated in the separate simulation environment (the Player/Stage robot simulation). When the robot moves through the maze, information from sensors about the position and barriers is provided from the robot simulation to the verification environment. You can see in Fig. 17 that the whole system consisting of two simulation environments works in the loop. The main output of the first phase is a claim whether the electronic controller (the robot controller) works correctly as specified or not. It is important because we have to be sure that the robot controller does not contain functional errors in the implementation. It is also important to point out that in this phase we acquire a set of verification scenarios (different mazes with different start and end positions for robot movements) that will also be used in the next phase. One verification run is represented by the robot moving through the maze from the start position to the end position.

The second phase consists of the verification using an FPGA with the verification scenarios obtained from the previous phase. It is guaranteed for these scenarios that if no artificial faults are injected into the system, the electronic part always behaves correctly. After a fault is injected, each of these scenarios is repeated (according to the number of injected faults). The result of this phase is a list of faults which causes a discrepancy on the output of the electronic controller for these specific verification scenarios. These faults will be examined in detail in the next phase where three possible outcomes can arise: (1) The output from the DUT and from the reference model is the same and an error did not appear. (2) The output is not identical but despite this, the robot has completed the mission (the robot reached the end position in the maze). (3) The output is not identical and at the same time, the mission was not accomplished. The last outcome is the most serious one and it will require a thorough analysis of the problem.

The analysis of the faults which affected badly the mechanical part is the task for the third phase. In this phase, we will examine the faults that caused the failure of the mission of the robot. This activity will be carried out manually, since it is necessary to run the required experiments repeatedly and to monitor the behavior of the mechanical part in the robot simulation as was described in the experimental part of this paper.

The generation of stimuli is a very important element in the proposed platform. In order to be able to check all working scenarios in functional verification and to achieve the highest possible coverage of all key functions in the verified circuit, a high-quality generator of inputs is needed. In our case, the generation aims at different mazes and a different starting and end positions of the movements of the robot. We also plan to use the generator for controlling the injection of faults (because now it is configured manually). We will generate signals that will drive the generation of faults and determine when and into which place a fault should be injected. The process of generating stimuli is described in the next Section 7.

7. Stimuli generation for the robot controller

We wanted to make the process of generating stimuli as universal as possible. Therefore, this approach is not limited only to the

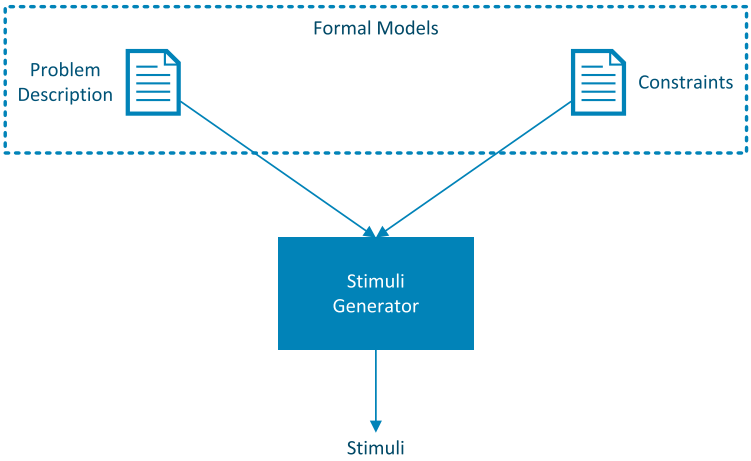


Fig. 19. The architecture of the stimuli generation.

Fig. 20. The parts of the problem description model.

robot controller, but it can be used also in other kinds of systems (the use case presented later in this paper demonstrates generating assembly programs for the processor). The architecture of the universal stimuli generator consists of two formal models (see Fig. 19): *description of the problem* and *constraints defined for the problem*. Each of these formal models is represented by one file with a specific purpose and a proprietary format.

The *Problem Description* model contains information about what has to be generated.

The second model *Constraints* contains restrictions and limitations for the problem described in the Problem Description model. This model defines valid combinations, the ordering and conditions for stimuli that are composed according to the constraints.

The core of the generation is the stimuli generator which takes these two formal models (files) as inputs and generates stimuli by 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.

7.1. The Problem Description model

The Problem Description model contains three basic parts for the problem definition (see Fig. 20): the substitute part, the variable part and the syntax part. Both of these models are defined by their own proprietary language.

The *Syntax part* defines the syntactic strings, one after another, which are needed to generate pseudo-random 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 syntactic strings. If they are not somewhere in the Syntax part, these variables and substitutes are ignored. The Syntax part represents static values, while the two remaining parts represent dynamic or changing values in the syntactic strings.

This part always starts with the keyword *syntax* followed by the "{" character, then contains *n*-lines of syntaxes and ends with the "}" character. The syntaxes can be easily grouped together for better clarity. The syntax of the Syntax part is the following:

```
syntax {
  synname1, synname2, ... {``generated
    word"}
  ...
}
```

The words synname1, synname2, etc. represent the name for each *generated word*. If it is needed to have some *generated word* with the same syntax, but with a different syntax name, it is possible to advantageously use the keyword *this* in the *generated word*.

The *Substitute part* defines all possible substitutes which will be pseudo-randomly replaced in 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 Substitute part is widely used in places where generating some specific words or phrases into the syntax is needed.

This part starts with the keyword *substitute* followed by the "{" character, then contains n-lines of substitutes and ends with the "}" character. The substitutes can be grouped together for better clarity. The syntax of the substitute part is the following:

```
substitute {
  repl1, repl2, ...{subs1|subs2|...}
  ...
}
```

The words repl1, repl2, etc. represent words that will be replaced in the *generated word*. The words subs1, subs2, etc. represent words that will be placed instead of words repl1, repl2, etc.

The *Variable part* defines the variables in a general sense. 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.

This part starts with the keyword *variable* followed by the "{" character, then contains n-lines of variables and ends with the "}" character. The syntax of the Variable part is:

```
variable {
  data_type varname
  ...
}
```

The data type can take one value from Table 2.

7.2. The constraints model

As mentioned earlier, constraints represent conditions and limitations for the generated stimuli. Constraints also ensure valid stimuli generation. This is essentially a limitation for data values, such as a variable cannot take certain values from the range of the data type, or restriction of dependency, such as some combination of variables cannot occur after the currently generated combination. The constraints model is unique for each system as well as the Problem Description model, therefore, various restrictions are applied to different systems.

Constraints are defined using a proprietary language and their syntax is like calling a function with parameters without the return value. The number of parameters is one or two because there was no need for complicated relationships between items in the Problem Description model. The syntax of a constraint is as follows:

$$constraintName(p_1, p_2, \ldots, p_n)$$

where n is the number of parameters ($n > 0$).

7.3. Stimuli Generator

The stimuli generator 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. The output from the generator is a set of lines that is valid for a defined problem.

The generation process starts with a random selection of one syntax from the syntax part. Based on the chosen syntax, the validity of all constraints that are defined for this syntax is tested sequentially. If all constraints are fulfilled, the syntax is sent to the output. If some of the constraints are not fulfilled, the generation process backtracks and a new substitute is chosen or a new value of a variable is generated. If some constraint is still not fulfilled, a new syntax is chosen and the process is repeated.

7.4. Maze generation

A maze represents one verification stimulus for the robot controller. Fig. 21 illustrates an idea of generating the mazes for the robot controller. This is an example that shows the function of the above-mentioned approach. The second example provided later in this paper is the generation of the assembly code for a processor. This use case is described in Section 8.3. The problem of generating the maze is defined as the generation of lines that are represented by the boolean array of a specific size. The constraints restrict the minimal width of the corridor of the maze, while the walls of the maze can be only rectangular and a room that has no path cannot appear in the maze. The result obtained by the generator is a sequence of rows that consists of zeroes or ones. Zeroes represent the corridors, ones represent the walls. This generated output may be further processed. In our case, this output is regenerated into a bitmap image representing the desired maze for the robot.

Table 2
Data types for variables.

Data type keyword	Min value	Max value	Note
BOOL	0	1	Boolean number
VAR8	0	255	8-bit unsigned integer
VAR16	0	65,535	16-bit unsigned integer
STR	STR0	STR29999	4–8 character long string

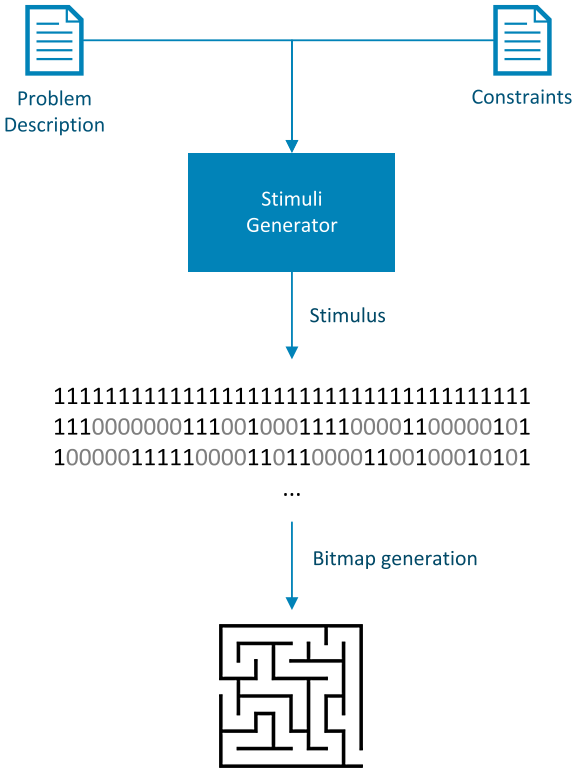


Fig. 21. An idea of generating a maze for the robot controller.

We analyzed this problem. There are a lot of approaches and algorithms for mazes generation [41], but none of them is suitable for the proposed universal process of generation. Therefore, maze generation is still in the design process and we are trying to find a suitable solution for our problem.

8. Use case – evaluation of processor

In our future work, we intend to concentrate on more complex mechanical systems controlled by their electronic controllers (not only just a robot in a maze). It is a well known fact that such electronic controllers are usually based on the use of processors to cover all the necessary functions (e. g. aerospace applications). Thus, in our research we decided to create the use case, where the electronic control unit is represented by a processor. Such an approach is described in this section. There already exist techniques for hardening processors against faults on the software level. This approach is called *Software Implemented Fault Tolerance*. An overview of these techniques is summarized in [42], a novel technique is also presented in [43]. Our idea is to evaluate the applicability of these techniques in the selected processor that will be placed into the FPGA and faults will be artificially injected into its architecture.

We decided to use the Codix RISC processor [44] of the Codasip company [45] as our test-case. Codix RISC is a 32-bit RISC processor with 7 stages of pipeline, 32 general purpose registers, 512 kB of the memory and 59 instructions. The architecture of the UVM-based verification environment is presented in the following subsection. For achieving a high level of coverage in functional verification, it is necessary to be able to generate a set of assembly programs for this processor. Generation of these programs by our universal generator is also described in the following subsections. The current state of our research is that we are able to generate the assembly programs and run functional verification for all these programs with correct results. The next step in our work will be applying the software fault tolerance to this processor and injecting faults.

8.1. Verification environment for processor

When referring to the first phase of our evaluation process presented in Fig. 18, we must create the functional verification environment for the processor and run the verification without injecting faults. The UVM-based verification environment is shown in Fig. 22 (implemented in SystemVerilog).

8.2. FPGA-based verification environment for processor

The second phase of the evaluation process (Fig. 18) is functional verification of the design implemented in the FPGA. Also, the fault injection into the FPGA takes place in this phase. For this purposes, the FPGA-based verification environment that is displayed in Fig. 23 is derived from the version created in the first phase. It should be noted that almost all UVM components are moved into the FPGA, except for the reference model and Scoreboard. Nevertheless, we aim at designing a consistent verification architecture in the FPGA too. Therefore, UVM Agents and their inbuilt components are just replaced by the HW Agents. We believe that consistent FPGA verification architecture is then easily understandable for verification engineers. The communication between the software and the hardware part of the verification environment is accomplished using a proprietary interface. More details about the components of both parts are provided in the subsequent subsections.

8.2.1. The software part of the verification environment

The main components of the software part are the Reference Model and Scoreboard. The Reference Model is in this specific case generated automatically from the high-level specification of the processor in the Codasip Studio [45], but of course, it can be implemented manually. Scoreboard compares results of the Reference Model to the results of DUT (received from the hardware part through the Output Wrapper component). In particular, we compare the content of memories and register fields when the specific assembly program is processed, and we continuously check data from the output ports. The Input Wrapper serves for sending programs (they are loaded to the processor and define its functionality) and input data.

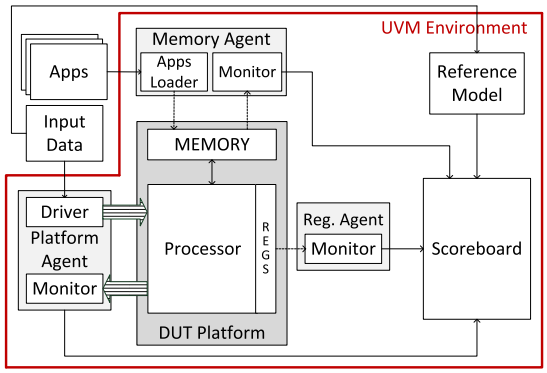


Fig. 22. The UVM-based verification environment for the processor.

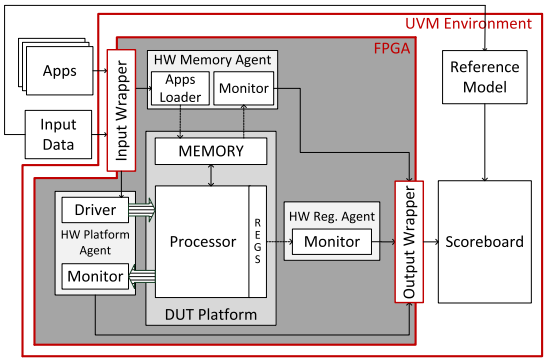


Fig. 23. The architecture of the FPGA-based verification environment for the processor.

8.2.2. The hardware part of the verification environment

Hardware Agents are similar to UVM Agents and their main components are Drivers and Monitors. Drivers drive the input ports of DUT and Monitors collect data from the output ports. In Fig. 23 you can see the Hardware Memory Agent, Hardware Register Agent and the Hardware Platform Agent. The hardware Memory Agent is connected to the main memory. It contains the Driver called the Application Loader that drives the loading of applications into the program part of the memory at the beginning of computation. The second component is the Monitor that takes an image of the memory at the end of the computation and sends it to the software Scoreboard for the comparison to the reference results. The Hardware Register Agent contains only the Monitor that takes an image of register fields at the end of the computation and sends it to the software Scoreboard. The Hardware Platform Agent is active during the whole computation; it contains the Driver that during the computation stimulates input ports of the processor with data and Monitor that sends the valid output data of the processor to the software Scoreboard.

8.3. Assembler stimuli generation for processor

Generation of the assembly code for a processor is one example of the use of the universal generation concept presented in the previous section. We designed the Problem Description model and the constraints model specifically for this test-case.

8.3.1. The Problem Description model for processor

The *syntax part* defines strings that we want to generate. We want to generate assembly 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 the following:

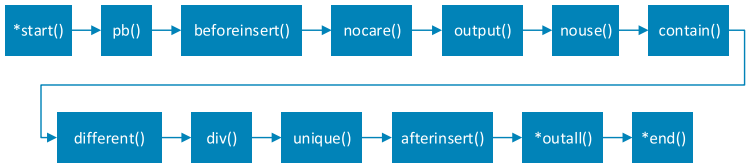


Fig. 24. The set of the constraints for generating the assembly code.

Table 3
The constraints for assembly code 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 that a substitute 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 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.

```
ori {‘‘dst = or srcl, 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. Here it is specified which substitutes will be replaced by a specific string. The example of one substitute is the following:

```
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.

8.3.2. The constraints model for processor

We have developed several constraints which solve typical problems of the assembly code generation. The set of constraints for the processors is shown in Fig. 24. The successive application of the constraints is also demonstrated. The constraints with star mark are evaluated only once during the generation, other constraints are evaluated for each instruction. The description of the constraints and their typical application in the assembly code generation is shown in Table 3.

8.3.3. Experimental results

As was already mentioned, the experiments were performed on the processor Codix RISC. For this processor, we have automatically generated the UVM-based functional verification environment and the verification process was running in the ModelSim simulator from Mentor Graphics [46].

The aim of the experiments is to achieve the maximum coverage of key system functions, because it guarantees the correctness of the system with respect to its specification. In the event that we will inject faults into the verified system, we can almost say with certainty that faulty system behavior is caused solely by these faults.

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

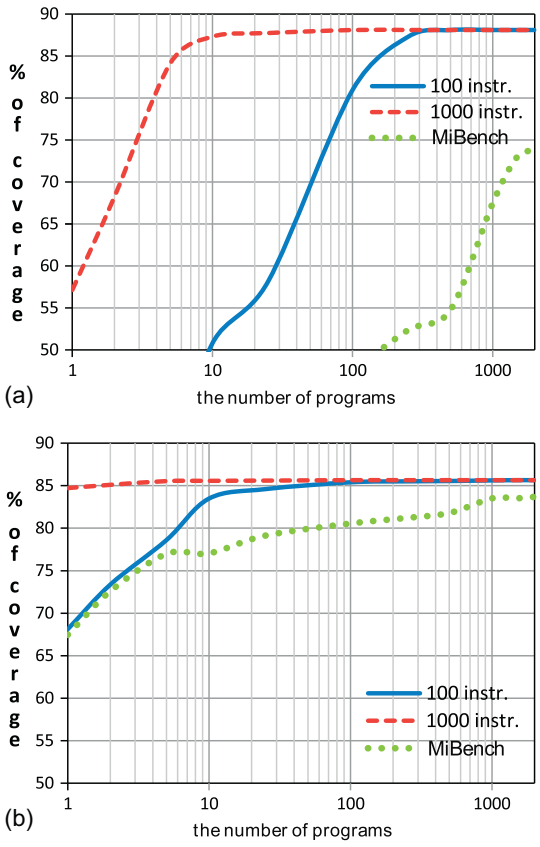


Fig. 25. Achieved (a) instruction coverage and (b) statement coverage in functional verification.

programs with 100 and 1000 instructions using the universal generator described before and we compared the results with the MiBench test program suite [47] 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. 25. *X*-axes of the graph 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 vectors. For 100 instructions, more programs were necessary than for 1000 instructions. In comparison with 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 it was higher for any number of programs than the coverage achieved by the MiBench test suite.

9. Conclusion and future work

In this paper, we introduced the evaluation platform for estimating the reliability of FPGA designs. As our research focuses on testing EM systems, we presented the experimental design which is composed of the mechanical robot and its electronic controller situated in the FPGA. The robot controller contains a variety of components. During the experiments, random faults were artificially injected into these components and we monitored the impact of these faults on the behavior of the robot in the simulation environment. These experiments showed that some faults have an impact on the behavior of the robot, and others do not. According to these results we were able to identify the parts/components of the robot controller that need to be hardened by some fault-tolerance techniques.

Two main goals were mentioned in Section 3, the first goal is to develop evaluation platform based on FPGA technology for checking the resilience of EM systems against faults. The presented work is the first step to achieve this goal, we performed preliminary experiments with EM system. The conclusions from these experiments are shown in Section 5.2 where the impact of faults in electronic controller of mechanical part was discussed. As for the second goal which aims at developing and verifying a new methodology for increasing fault-tolerance qualities of EM systems, the main idea how to achieve it was presented in Section 6. The foundations for the proposed methodology are also presented as the conclusions of the performed experiments.

In addition, we recognized from the experiments that some kind of automation is unavoidable in our future experiments, especially in the early phases of testing. The reason is that monitoring the behavior of the system in simulation is very time-demanding. Therefore, we have already prepared an innovative extension of our platform – interconnection of fault injection and functional verification environment with an advanced stimuli generation. Using this approach we will be able to automatically verify an EM system during the fault injection. Automation is achieved by comparing the outputs of the verified system to the reference model that is in our case represented by the same design but without injected faults.

Acknowledgments

The research was supported by the following European projects. This work was supported by the following projects: EU COST Action IC1103 – MEDIAN – "Manufacturable and Dependable Multicore Architectures at Nanoscale", project IT4Innovations Centre of Excellence (ED1.1.00/02.0070), National COST LD12036 – "Methodologies for Fault Tolerant Systems Design Development, Implementation and Verification" and BUT project FIT-S-14-2297.

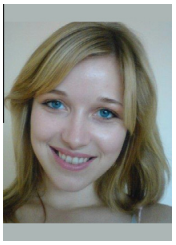
References

[1] S. Cutts, A collaborative approach to the more electric aircraft, in: International Conference on Power Electronics, Machines and Drives, 2002 (Conf. Publ. No. 487), 2002, pp. 223–228, http://dx.doi.org/10.1049/cp:20020118.
[2] J. Bennett, A. Jack, B. Mecrow, D. Atkinson, C. Sewell, G. Mason, Fault-tolerant control architecture for an electrical actuator, in: 35th Annual Power Electronics Specialists Conference, 2004, PESC 04, 2004, vol. 6, IEEE, 2004, pp. 4371–4377, http://dx.doi.org/10.1109/PESC.2004.1354773.
[3] G. Leen, D. Heffernan, Expanding automotive electronic systems, Computer 35 (1) (2002) 88–93, http://dx.doi.org/10.1109/2.976923.
[4] M. Straka, J. Kastil, Z. Kotasek, L. Miulka, Fault tolerant system design and SEU injection based testing, Microprocess. Microsyst. 2013 (37) (2013) 155–173.
[5] XILINX, FPGA, November 2014 <http://www.xilinx.com/fpga/index.htm>.
[6] F. Piltan, N. Sulaiman, M. Marhaban, A. Nowzary, M. Tohidian, Design of FPGA-based sliding mode controller for robot manipulator, Int. J. Robot. Autom. (IJRA) 2 (3) (2011) 173–194.
[7] U.D. Meshram, R. Harkare, FPGA based five axis robot arm controller, in: IEEE Conference, 2005, pp. 3520–3525.
[8] N. Instruments, Powertrain Controls (May 2015) <http://sine.ni.com/ind-app/app/app/p/id/app-71/lang/cs>.
[9] M. Ceschia, M. Violante, M. Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, A. Candelori, Identification and classification of single-event upsets in the configuration memory of SRAM-based FPGAs, IEEE Trans. Nucl. Sci. 50 (6) (2003) 2088–2094.
[10] J.A. Cheatham, J.M. Emmert, S. Baumgart, A Survey of Fault Tolerant Methodologies for FPGAs, vol. 11, ACM, New York, NY, USA, 2006. pp. 501–533.
[11] F.L. Kastensmidt, R. Reis, Fault-Tolerance Techniques for SRAM-Based FPGAs, vol. 32, Springer, 2007.
[12] F.L. Kastensmidt, G. Neuberger, L. Carro, R. Reis, Designing and testing fault-tolerant techniques for SRAM-based FPGAs, in: Proceedings of the 1st conference on Computing frontiers, ACM, 2004, pp. 419–432.
[13] XILINX, Partial Reconfiguration User Guide.
[14] C. Bolchini, A. Miele, M.D. Santambrogio, TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGAs, in: DFT '07: Proceedings of the 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems, IEEE Computer Society, Washington, DC, USA, 2007, pp. 87–95.
[15] L. Sterpone, M. Aguirre, J. Tombs, H. Guzmán-Miranda, On the design of tunable fault tolerant circuits on SRAM-based FPGAs for safety critical applications, in: DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe, ACM, New York, NY, USA, 2008, pp. 336–341.
[16] R. Oliveira, A. Jagirdar, T.J. Chakraborty, A TMR scheme for SEU mitigation in scan flip-flops, in: ISQED '07: Proceedings of the 8th International Symposium on Quality Electronic Design, IEEE Computer Society, Washington, DC, USA, 2007, pp. 905–910.
[17] C. Bernardeschi, L. Cassano, A. Domenici, L. Sterpone, Accurate simulation of SEUs in the configuration memory of SRAM-based FPGAs, in: IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2012, IEEE, 2012, pp. 115–120.
[18] S. Rudrakshi, V. Midasala, S. Bhavanam, Implementation of FPGA based fault injection tool (FITO) for testing fault tolerant designs, IACSIT Int. J. Eng. Technol. 4 (5) (2012) 522–526.
[19] M. Alderighi, S. D'Angelo, M. Mancini, G.R. Sechi, A fault injection tool for SRAM-based FPGAs, in: 9th On-Line Testing Symposium, 2003, IOLTS 2003, IEEE, IEEE, 2003, pp. 129–133.
[20] M. Alderighi, F. Casini, S. d'Angelo, M. Mancini, S. Pastore, G.R. Sechi, Evaluation of single event upset mitigation schemes for SRAM based FPGAs using the flipper fault injection platform, in: 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems, 2007, DFT'07, IEEE, 2007, pp. 105–113.
[21] C. López-Ongil, M. Garcia-Valderas, M. Portela-Garća, L. Entrena, Autonomous fault emulation: a new FPGA-based acceleration system for hardness evaluation, IEEE Trans. Nucl. Sci. 54 (1) (2007) 252–261.
[22] M. Straka, J. Kastil, Z. Kotasek, SEU simulation framework for xilinx fpga: First step towards testing fault tolerant systems, in: 14th EUROMICRO Conference on Digital System Design, IEEE Computer Society, 2011, pp. 223–230.
[23] T. Kropf, Introduction to Formal Hardware Verification, Springer, 1999 <http://books.google.cz/books?id=p3xSw3AllToC>.
[24] A. Meyer, Principles of Functional Verification, Elsevier Science, 2003 <http://books.google.cz/books?id=qaIiX3hYWL4C>.
[25] M. George, O. Ait Mohamed, Performance analysis of constraint solvers for coverage directed test generation, in: 2011 International Conference on Microelectronics (ICM), 2011, pp. 1–5, http://dx.doi.org/10.1109/ICM.2011.6177404.
[26] D. Gohel, Pure SV verification environment methodology for asic verification 5 (2014) 770–775.

[27] S. Fine, A. Ziv, Coverage directed test generation for functional verification using bayesian networks, in: Proceedings of the Design Automation Conference, 2003, 2003, pp. 286–291, http://dx.doi.org/10.1109/DAC.2003.1219010.
[28] L. Kotthoff, Constraint Solvers: An Empirical Evaluation of Design Decisions, ArXiv e-printsarXiv:1002.0134.
[29] V. Kumar, Algorithms for constraint satisfaction problems: a survey, AI Magaz. 13 (1) (1992) 32–44.
[30] N. Rollins, M. Fuller, M. Wirthlin, A comparison of fault-tolerant memories in SRAM-based FPGAs, in: 2010 IEEE Aerospace Conference, 2010, pp. 1–12, http://dx.doi.org/10.1109/AERO.2010.5446661.
[31] M. Naseer, P. Sharma, R. Kshirsagar, Fault tolerance in FPGA architecture using hardware controller – a design approach, in: International Conference on Advances in Recent Technologies in Communication and Computing, 2009, ARTCom '09, 2009, pp. 906–908, 2009, http://dx.doi.org/10.1109/ARTCom.236.
[32] L. Frigerio, F. Salice, Ram-based fault tolerant state machines for FPGAs, in: 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems, 2007, DFT '07, 2007, pp. 312–320, http://dx.doi.org/10.1109/DFT.2007.33.
[33] B. Gerkey, R.T. Vaughan, A. Howard, The player/stage project: tools for multi-robot and distributed sensor systems, in: Proceedings of the 11th International Conference on Advanced Robotics, vol. 1, 2003, pp. 317–323.
[34] Z. Vasicek, FITkit, April 2014 <http://www.fit.vutbr.cz/FITkit>.
[35] J. Podivinsky, M. Simkova, Z. Kotasek, Complex control system for testing fault-tolerance methodologies, in: Proceedings of The Third Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale (MEDIAN 2014), COST, European Cooperation in Science and Technology, 2014, pp. 24–27.
[36] OPENCORES, Wishbone B4: WISHBONE System-on-Chip (SoC) Interconnection Architecture Portable IP Cores, April 2014 <http://cdn.opencores.org/downloads/wbspecb4.pdf>.
[37] XILINX, Xst User Guide.
[38] N. Dorairaj, E. Shiflet, M. Goosman, Planahead software as a platform for partial reconfiguration, Xcell J. 55 (68–71) (2005) 84.
[39] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, B. Hutchings, Rapid prototyping tools for FPGA designs: Rapidsmith, in: 2010 International Conference on Field-Programmable Technology (FPT), 2010, pp. 353–356, http://dx.doi.org/10.1109/FPT.2010.5681429.
[40] M. Simkova, O. Lengal, M. Kajan, Haven: An Open Framework For FPGA-Accelerated Functional Verification of Hardware, Tech. rep., 2011 <http://www.fit.vutbr.cz/research/viewpub.php.en?id=9739>.
[41] P.W., Maze Algorithms, 1996 <http://www.astrolog.org/labyrnth/algrithm.htm>.
[42] O. Goloubeva, M. Rebaudengo, M.S. Reorda, M. Violante, Software-Implemented Hardware Fault Tolerance, Springer Science+Business Media, LLC, New York, 2006. p. 224.
[43] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, D.I. August, Swift: software implemented fault tolerance, in: Proceedings of the International Symposium on Code Generation and Optimization, IEEE Computer Society, 2005, pp. 243–254.
[44] Codasip, Codix RISC, November 2014 <https://www.codasip.com/products/codix-risc/>.
[45] Codasip, Codasip Framework, November 2014 <http://www.codasip.com>.
[46] U. Hatnik, S. Altmann, Using modelsim, matlab/simulink and ns for simulation of distributed systems, in: International Conference on Parallel Computing in Electrical Engineering, 2004, PARELEC 2004, 2004, pp. 114–119, http://dx.doi.org/10.1109/PCEE.2004.74.
[47] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, Mibench: a free, commercially representative embedded benchmark suite, in: 2001 IEEE International Workshop Proceedings of the Workload Characterization, 2001, WWC-4, WWC '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 3–14, http://dx.doi.org/10.1109/WWC.2001.15.

Jakub Podivinsky was born in 1989. In 2013 he graduated (M.Sc.) at the Department of Computers Systems of the Faculty of Information Technology, Brno University of Technology. In 2013 he started his Ph.D. studies at the Department of Computers Systems. His scientific research is focused on evaluation quality of fault tolerant systems and FPGA-based functional verification of digital systems.

Ondrej Cekan was born in 1989. In 2013 he graduated (M.Sc.) at the Department of Computers Systems of the Faculty of Information Technology, Brno University of Technology. In 2013 he started his Ph.D. studies at the Department of Computers Systems. His scientific research is focused on functional verification and stimuli generation.



Marcela Simkova was born in 1987. In 2011 she graduated (M.Sc.) at the Department of Computers Systems of the Faculty of Information Technology, Brno University of Technology. In 2011 she started her Ph.D. studies at the same university. Her scientific research is focused on optimization of UVM-based functional verification, automated verification of processors and fault-tolerant system design.

Zdenek Kotasek was born in 1947. He received his M.Sc. and Ph.D. degrees (in 1969 and 1991) from Brno University of Technology (BUT), both in computer science. Between 1969 and 2001, he worked at Department of Computer Science of the Faculty of Electrical Engineering and Computer Science, since 2002 at the Department of Computer Systems (DCSY) of the Faculty of Information Technology, both at BUT. He is an Associate Professor at BUT since 2000 and the head of the DCSY (since 2005). His research interests include digital circuit diagnostics and testing, testability analysis and design and synthesis for testability and reliability, fault tolerant system design. He is an IEEE member (since 2003).

Paper II

Software Fault Tolerance: the Evaluation by Functional Verification

ČEKAN Ondřej, PODIVÍNSKÝ Jakub, KOTÁSEK Zdeněk
In: *Proceedings of the 18th Euromicro Conference on Digital Systems Design.* Funchal: IEEE Computer Society, 2015, pp. 284-287. ISBN 978-1-4673-8035-5.

2015 Euromicro Conference on Digital System Design

Software Fault Tolerance: the Evaluation by Functional Verification

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

***Abstract*—The aim of this paper is to present a new approach in evaluating *Software Fault Tolerance* (SFT) methodologies. It is the way on how to ensure fault tolerance without any additional hardware as is common in frequently used *Triple Modular Redundancy* (TMR). As our research is focused on electro-mechanical systems which are commonly driven by processors or *Multi Processors Systems on Chip* (MPSoC) we decided to use the soft-core processor running on *Field Programmable Gate Array* (FPGA) as our experimental platform. The new approach uses *Functional Verification* for automation of the evaluation process. The functional verification environment is one of the important parts of the presented evaluation platform architecture. Programs generation for a processor, where SFT is applied, is also important. Experiments with the programs generator and fault injection are presented and goals for future work are identified on that basis.**

Keywords*—*Software Fault Tolerance, SFT, Processor, Fault Injection, Electro-mechanical Systems, Functional Verification.

I. Introduction

In several areas, various mechanical applications are controlled by their electronic controllers, for example, medical equipments or aerospace systems. This systems are usually called *Electro-mechanical* or *Cyber-physical Systems* [1]. Electronic controllers are often implemented by processors or, in the case of complex systems, *Multi-Processors Systems-on-Chip* (MPSoCs) are used. Controllers can be also implemented by *Field Programmable Gate Array*s (FPGAs) [2] which are becoming wider used.

Only FPGAs are in the scope of our research, as we are focused on fault-tolerant systems implemented in FPGA and their evaluation by fault injection. FPGAs are composed of *Configurable Logic Blocks* (CLBs) that are interconnected by a programmable interconnection net. Every CLB consists of *Look-Up Tables* (LUTs) that realize the logic function, a multiplexer and a flip-flop. The configuration of CLBs and of the interconnection net is stored in SRAM memory. The problem from the reliability point of view is that FPGAs are sensitive to faults caused by charged particles [3]. These particles can induce an inversion of a bit in the configuration SRAM memory of an FPGA (or directly to its internal flip-flops) and this may lead to a change in its behaviour. This event is called the *Single Event Upset* (SEU).

The evaluation platform for testing fault-tolerance methodologies based on FPGAs in the context of electro-mechanical applications was presented in our previous paper [4]. This evaluation platform is composed of a robot, which represents a mechanical part, and its electronic controller. It is possible to apply various fault-tolerance methodologies on this controller and evaluate the impact of injected faults, not only on the electronic part, but also on the behaviour of the mechanical robot. In our future work, we intend to use the processor or MPSoC implemented in FPGA as an experimental electronic controller. This paper is first step to achieve this goal. The evaluation platform for testing fault-tolerance methodologies targeted to processors, especially the software implemented fault-tolerance, is presented in this paper.

The paper is organized as follows. The related work connected to the processor reliability is summarized in Section II. The goals of our research, using functional verification for evaluating software fault tolerance can be found in Section III. A detailed description of the generation process of programs for a processor implemented as software fault tolerant is available in Section IV. Results of experiments with generated fault tolerant programs are in Section V. Future work connected to our research can be found in Section VI. Finally, the results are summarized in Section VII.

II. Related Work

In general, there are two ways on how to harden processors against faults: (1) *Hardware Fault Tolerance* and (2) *Software Fault Tolerance*, also known as *Software Implemented Hardware Fault Tolerance*. A fault-tolerant processor architecture is presented in [5]. This architecture is based on inserting a checker unit in front of the processor commit stage. The checker unit re-executes both computation and memory/register field reads. Whenever an error is detected, the checker is assumed to be fully reliable and fixes the error, then commits results, flushes the processor and restarts it at the next instruction. Special techniques for soft-core processors running on FPGA are also presented in the literature. In [6], an overview of commonly used fault tolerant techniques based on FPGA which can be used in the context of processors is presented. The authors discuss *Triple Modular Redundancy* (TMR) architecture, *Duplex architecture* and the use of *Partial Dynamic Reconfiguration* (PDR). The use of PDR is also presented in [7] and a fault tolerant approach on *Look-up Table* (LUT) level is presented and the technique is evaluated on a RISC processor. As mentioned above, hardware redundancy is not in the scope of this paper and we focus on the software fault tolerance.

A. Software Fault Tolerance

In our research, we are focusing on transient faults caused by SEUs. Transient faults are errors that occur unpredictably due to charged particles or electro-magnetic interferences. We did not solve these problems classically by additional

978-1-4673-8035-5/15 $31.00 © 2015 IEEE
DOI 10.1109/DSD.2015.107

hardware (hardware redundancy), but we used techniques that ensure correct behaviour by the software (time or information redundancy). One of the possible ways that can be used in order to deal with such errors, is *Software Fault Tolerance* (SFT). It is sometimes used as extra protection of the software. SFT is a commonly used technique which ensures the continuous availability of service while maintaining the desired performance and safety of the software in case faults. Fault Tolerance (FT) is generally very important in safety-critical applications. SFT can be divided into several levels that ensure availability and data consistency of the application.

In the case that we do not count software without fault tolerance, four levels [8] of SFT exist:

1) Automatic detection and restart,
2) Level 1 + checkpoints, logs and initial state recovery,
3) Level 2 + persistent data recovery,
4) Continuous operation without any interruption.

Level 1 uses mechanisms that can detect an error in the software. This level of SFT utilizes Error Control Codes (ECC) through which the software is able to detect errors and protect itself against faults. *Level 2* adds to the previous level the possibility of storing the internal state of the application process. Storing takes place in the form of periodic checkpoints, where the critical volatile data are saved. *Level 3* uses an additional backup disk on which the persistent data of the application is replicated. The backup data are consistent with data in the application process. In case of error detection, the application is recovered from the backup disk.

Level 4 is the last level of SFT in which an interruption does not occur after an error and recovery is masked. At this level, there are frequently used replication techniques such as process replication, N-version software or information and time redundancy. This level of SFT is generally referred under the term *software redundancy* [9]. This is typical for safety-critical real-time applications. Availability and data consistency of the application are on the highest possible value for this level than for previous levels. Software redundancy is typically based on additional instructions which are added extra into the original application. There are two possibilities on how the software redundancy can be performed:

- by hardening the data,
- by hardening the control flow.

The first method uses operation and information redundancy which leads to duplication of instructions, procedures or programs. There are three general characteristics for this method. First, the memory space for the original program is at least two times larger in the modified program. Second, the computation time of modified program is two times slower than the original program. Third, the programmer must follow specific rules for programming data structures and statements.

The second method uses special techniques for hardening a microprocessor-based system against errors of control flow. These errors cause changes in instructions (it does not modify only data as previous errors) and thus unexpected fetch and execution of the instruction by the processor. In [9], many techniques for tackling these errors are described. Because SEUs invade memory elements, we focus on the first method of hardening.

In the following sections, we will propose our approach based on time and information redundancy through triplication of instructions with the same values stored in different registers.

III. The Goals of The Research

The main goal of research presented in this paper is the evaluation of *Software Fault Tolerance* on a processor running on an FPGA, also called a soft-core processor. Our plan is an automatic generation of programs for a processor where software fault tolerance is applied and then the impact of fault by fault injection into memory cells of an FPGA is evaluated. Precise evaluation is not possible without using some kind of automation which is the main reason why we decided to use a technique called *Functional Verification*.

The functional verification [10] is the process of verifying that a model of the system, also called *Device Under Test* (DUT), complies with the specification by monitoring inputs and outputs in a simulation. Moreover, the DUT outputs are compared to the outputs of the reference model that is typically implemented by a verification engineer or a designer that did not implement the DUT. On the basis of comparing the outputs a discrepancy between the two models can be detected and thus an error in the systems can be discovered. In order to be able to inject faults into the FPGA while performing functional verification, we must carry out the verification directly in the FPGA (not in the simulation as is usual). Advantageously we can use and modify hardware accelerated verification that uses an FPGA as the acceleration board. An example of such an accelerator, framework HAVEN [10] can be noted.

The main principle of using functional verification of a processor running on FPGA for evaluating software fault tolerance is shown in Figure 1. *Apps Generator* is needed for generating fault tolerant programs for a processor. Not only is the quality of fault tolerance required, but a functional coverage which is ensured by generating a sufficiently large set of programs too. *DUT Platform* is the main component which contains a processor where the generated programs are executed. When we need *Fault Injection* for fault simulation it is possible to run DUT platform on an FPGA, but for the functional coverage measures of generated sets of programs an FPGA is not required. *Reference Model* and *Scoreboard* for detecting differences in processor output can also be identified.

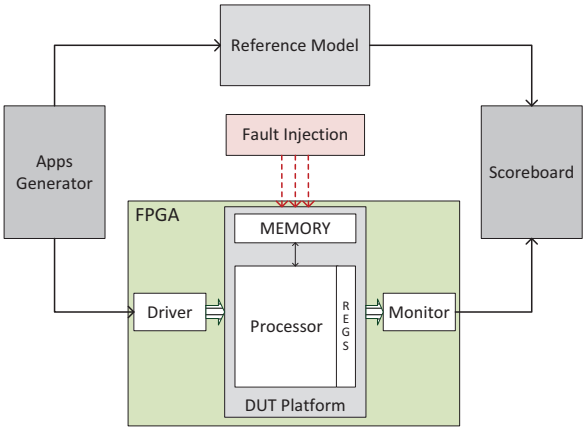


Fig. 1. The overview scheme of using functional verification for evaluating software fault tolerance.

IV. Fault-tolerance Programs Generation

In Section II, we chose the method of SFT for hardening the data against SEU errors. The idea is based on information redundancy which is added into the assembly instruction level. In our previous work [4], our approach of universal stimuli generation that we use in this work in a convenient way was presented. Our stimuli generator is used for the purpose of generating fault-tolerant assembly programs which are immune against SEU errors. We use a modification of the technique of instruction duplication which is based on triplication of instructions. Although Triple Modular Redundancy (TMR) is used predominantly in hardware, we used principles of TMR in software. We have implemented software Triple Instructional Redundancy (TIR) which is an analogy of TMR in the hardware. Verifying fault-tolerance effectiveness of programs is performed by an injector for simulation of SEU errors which modifies data using suitably positioned instructions. To our best knowledge, practical implementation of TMR in software which concurrently verifies fault-tolerance using simulation of SEU errors in the software does not exist in the literature. In the following text, the whole process of generating TIR programs, including the injection, is presented.

A. Triple Instruction Redundancy

TIR has the same scheme (see Figure 2a) as TMR, but it operates on the software level of program instructions. In the figure, I1-I3 (instruction elements) are instances of the same instruction, V1-V3 are majority voters for the results obtained from each instruction and interconnections determine the binds between instruction results and voters for evaluation of the majority. Our focus is on fault-tolerance in the registers of the processor, therefore, three voters are needed for conservation of error-less behaviour. In every cycle of the processor, an SEU may occur, therefore, three voters have to be placed after each instruction element. If we focus on fault-tolerance in data memory, only one voter can be used, because voters typically work with register data which cannot be damaged in data memory orientation.

In Figure 2b, the example of TIR is shown. Every I1-I3 element of TIR is the carrier of value (register) for the instruction result. Each instruction I1-I3 cannot use all registers, because they would rewrite their results. As can be seen, the I1 element has a wrong value after the instruction execution. This TIR mechanism is able to recover from SEU error. The voter on the instruction level performs a comparison operation which writes into the same register as the instruction element.

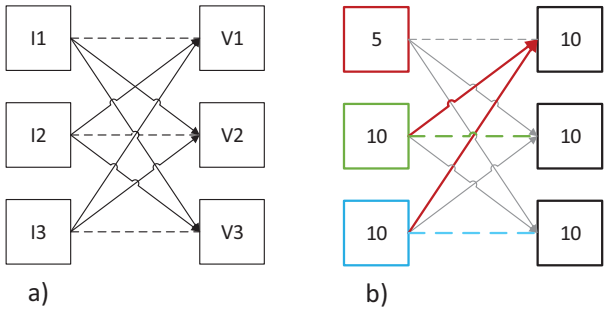


Fig. 2. a) Principle of TIR architecture with 3 voters and b) its example.

B. Simulation of SEUs Injection

Simulation of SEU errors in software is performed by inserting an instruction in a specific position in the program. Through instruction generation without TIR mechanism we are able to simulate SEUs that may occur between the voter and instruction element or between instruction element and the voter. The main idea is to generate an instruction which writes into a register after the execution of instruction elements or voters. The result of this injection is the rewriting of exactly one register. This operation is equivalent to the single event upset. These instructions can be generated in the program with a certain probability, therefore, more SEUs may be injected into the program in different positions. The example of simulation of the SEU injection in an assembly RISC program between instruction elements and voters is shown in Figure 3.

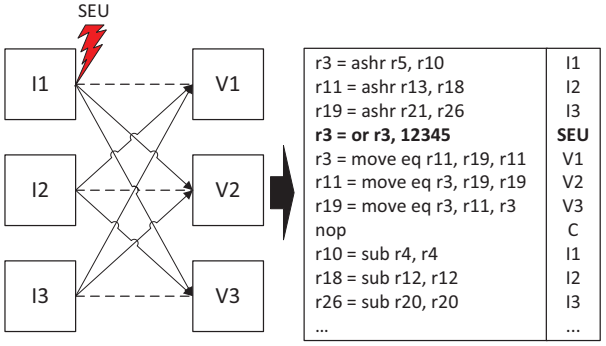


Fig. 3. Example of simulation of the SEU injection.

C. TIR Programs Automatic Generation

An automatic assembly code generation is based on the universal principle of generation presented in [4]. The generator is based on two specific input structures which define a format of generated data (Problem Description) and restrictions and conditions (Constraints) and on how the data have to be generated in order to be valid for a specific system. For the TIR definition, each syntax of instruction of the processor has to be triplicated in its definition in the Problem Description structure and has to be also followed by three comparison instructions representing the voters. The instructions which do not write any values into registers, do not have to be secured by TIR and their definition is described only by their real syntaxes. Constraints structure contains commands which ensure the valid generating of jumps and labels, memory access, latencies, probabilities etc. For the TIR definition, one additional constraint has to be created. This constraint cares about the distribution of registers between TIR instances. The process of program code generation starts with selecting an instruction. Then the constraints for the instruction are evaluated on the validity. If all constraints are satisfied, six instructions are written to output in the case of TIR with three voters. Various instructions put together the final fault-tolerant program for the processor. SEU injection is simulated by generating the instruction which contains only its real syntax. The whole process of TIR program generation is illustrated by a block diagram in Figure 4. We are able to generate concurrently programs with and without a software fault-tolerance TIR technique between them there is an equivalence.

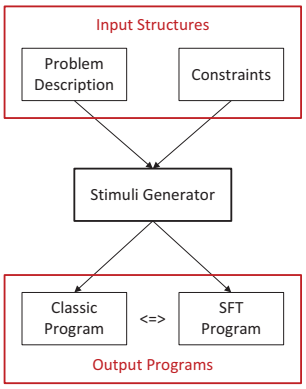


Fig. 4. The process of TIR program generation.

V. Experiments and Results

According to the principle presented in Section IV-C, we have defined both input structures (problem description and constraints) which are needed for the automatic generation of TIR programs for the Codix RISC processor. The most interesting experiments from the perspective of evaluating software fault tolerance are experiments with fault injection. The principle of fault injection without using a real FPGA is presented in Section IV-B. From measurement, we decided to use TIR programs with 100 instructions per program for our next experiment. The same number of TIR programs was generated with n inserted faults where $0 < n \leq 13$. It means, that we have n sets of TIR programs and the difference between these sets is the number of injected faults into each of the programs. This allow us to perform statistical experiments where probability of failure is evaluated for each n. The maximal value of n (13) was obtained experimentally, because for a higher number of inserted faults the same probability was achieved (100% of failure).

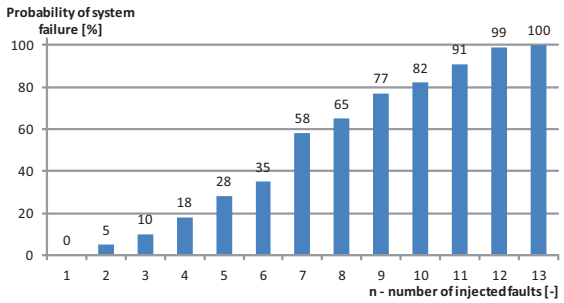


Fig. 5. Bar graph showing the probability of system failure if n faults are injected.

Measured statistical data are shown in Figure 5. It contains n bars and each of them shows the probability of system failure when n faults are injected. It can be assumed, based on principles of TIR, that only one injected fault has no apparent effect on processor output. This is confirmed by our experiments because the Figure show 0% of single faults that lead to the failure. The main contribution of these experiments is a statement that functional verification of a processor can be effectively used for the evaluation of software fault tolerance. The conclusion forms the root for our future work base on the use of functional verification with a processor on a real FPGA.

VI. Future Work

There are two main goals in our future work. The short term goal is the direct continuation of this paper which means the use of the designed architecture for testing Software Fault Tolerance on a real FPGA by the fault injection. It was mentioned in Section I that our main research goal is focused on testing fault tolerance methodologies based on FPGA in the context of electro-mechanical systems [4]. This is our long term goal and in the context of this paper means connecting processor and software fault tolerance with electro-mechanical systems.

VII. Conclusion

In this paper, a new approach to the evaluation of *Software Fault Tolerance* which is based on functional verification of processor is described. The main principle of SFT is shown. We have focused on software redundancy. We presented the principle of automatic generation of SFT programs. These programs were evaluated by using the simulation-based variant and measured statistics were shown in graph. Our future work is based on processor implementation on real FPGA. This paper belongs to our long term goals which are focused on evaluation of FPGA-based fault tolerant methodologies in the context of electro-mechanical applications. In this area, we plan to use a processor implemented in FPGA as an electronic controller of the mechanical system and evaluate SFT in a real application.

Acknowledgment

This work was supported by the following projects: EU COST Action IC1103 "MEDIAN", National COST LD12036, project Centre of excellence IT4Innovations (ED1.1.00/02.0070), and BUT project FIT-S-14-2297.

References

[1] S. Khaitan and J. McCalley, "Design techniques and applications of cyberphysical systems: A survey," *Systems Journal, IEEE*, vol. PP, no. 99, pp. 1–16, 2014.

[2] D. Macii, M. Avancini, L. Benciolini, S. Dalpez, M. Corra, and R. Passerone, "Design of a redundant fpga-based safety system for railroad vehicles," in *17th Euromicro Conference on Digital System Design (DSD), 2014*, Aug 2014, pp. 683–686.

[3] M. Ceschia, M. Violante, M. Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, and A. Candelori, "Identification and classification of single-event upsets in the configuration memory of SRAM-based FPGAs," *Nuclear Science, IEEE Transactions on*, vol. 50, no. 6, pp. 2088–2094, 2003.

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

[5] A. Bouajila, T. Sommer, J. Zeppenfeld, W. Stechele, and A. Herkersdorf, "A fault-tolerant processor architecture," in *22nd International Conference on Architecture of Computing Systems (ARCS), 2009*, March 2009, pp. 1–6.

[6] J. S. Patel and D. H. Shah, "Different types of fault tolerant techniques of softcore processor," *I-journal*, 2014.

[7] A. Vavousis, A. Apostolakis, and M. Psarakis, "A fault tolerant approach for fpga embedded processors based on runtime partial reconfiguration," *Journal of Electronic Testing*, vol. 29, no. 6, pp. 805–823, 2013.

[8] Y. Huang and C. Kintala, "10 software fault tolerance in the application layer," 1995.

[9] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[10] M. Šimková, *Hardware Accelerated Functional Verification*. Lambert Academic Publishing, 2011.

Paper III

Verification of Robot Controller for Evaluating Impacts of Faults in Electro-mechanical Systems

PODIVÍNSKÝ Jakub, ČEKAN Ondřej, LOJDA Jakub, KOTÁSEK Zdeněk

In: *Proceedings of the 19th Euromicro Conference on Digital Systems Design.* Limassol: IEEE Computer Society, 2016, pp. 487-494. ISBN 978-1-5090-2817-7.

2016 Euromicro Conference on Digital System Design

Verification of Robot Controller for Evaluating Impacts of Faults in Electro-mechanical Systems

Jakub Podivinsky, Ondrej Cekan, Jakub Lojda, Zdenek Kotasek
Faculty of Information Technology, Brno University of Technology
Bozetechnova 2, 612 66 Brno, Czech Republic
Tel.: +420 54114-{1361, 1361, 1360, 1223}
Email: {ipodivinsky, icekan, ilojda, kotasek}@fit.vutbr.cz

***Abstract*—Functional verification is a modern approach to verifying that a digital system complies with its specification. The verification environment for functional verification of robot controller which searches path for the robot through a maze is presented in this paper. This verification environment is designed according to UVM (Universal Verification Methodology) principles. As an interesting feature of the verification environment we see the use of a mechanical part (robot in a maze) simulation. The article describes the use of the verification environment for evaluating impacts of faults in electro-mechanical systems. It will serve as a tool for automating the fault tolerance evaluation of electro-mechanical systems and together with the fault injector will form the basis of the verification platform in the future. The experimental results gained from the verification process are also presented in the paper.**

Keywords*—Functional Verification, Robot Controller, Electro-mechanical Systems, Fault Tolerance, Maze Generation.

I. Introduction

Digital systems play an important role in our everyday lives. They are widely used in industry, medicine and other safety critical sectors. Not only the loss of a huge amount of money, but also the loss of human lives may occur in case of their failure. The current trend is that the complexity of digital systems rises, which leads to an increased susceptibility to faults. It is possible to specify two main sources of faults [1]: 1) *Design faults* (bugs) are always the consequence of an incorrect design, an ambiguous specification or misinterpretation of the specification and 2) *Hardware/physical faults* (defects) which arise during manufacturing or during system operation.

The approach which deals with design faults is called *functional verification* [2] which currently has an irreplaceable position in the development cycle of digital systems. Functional verification checks whether a hardware system satisfies a given specification. The main purpose is to find as many design faults as possible before the system is deployed. The main principle of functional verification is to compare the outputs of verified circuits with those of the reference model. Different coverage metrics are defined in order to assess that the design has been adequately exercised. These include code coverage and functional coverage. *Code coverage* gives information about how many lines and how many times expressions and branches are executed. This coverage is collected by the simulation tool. *Functional coverage* is defined by the user. The user defines the coverage points for the functions to be covered in a verified circuit (DUT - Design Under Test) and it is completely under user control. Moreover, standard languages, methodologies and libraries were defined for functional verification. The most commonly known are the SystemVerilog IEEE language standard, Universal Verification Methodology (UVM) [3] and the open-source UVM library (with all the basic components of verification environments).

The techniques called *Fault avoidance* or *Fault tolerance* [4] deal with the second type of faults called hardware/physical faults. *Fault avoidance* is mainly obtained by the use of radiation hardened technologies, improved design of storage elements or asynchronous circuits. *Fault tolerance* is the ability of a system to continue performing its correct function even in the presence of unexpected faults. There have been many fault-tolerant methodologies inclined, among others, to *Field Programmable Gate Arrays* (FPGAs) developed and new ones are under investigation [5], because FPGAs are becoming more popular due to their flexibility and reconfigurability. The second reason why so many techniques are inclined to FPGAs is their sensitivity to faults and ability to be reconfigured in the case of fault occurrence. FPGAs are composed of configurable logic blocks [6] which are connected by programmable interconnection. The configuration is stored as a *bitstream* in SRAM memory. The problem is that FPGAs are quite sensitive to faults caused by charged particles [7]. This particle can induce inversion of a bit in bitstream and this may lead to a change in its behaviour. This event is called *Single Event Upset* (SEU).

It is important to test and evaluate these techniques. Various approaches to the evaluation of fault tolerance exist, some of them are performed on a theoretical level, for example, a simulation method for SEU emulation is presented in [8]. Another approach is in the use of fault injection directly to the design implemented in FPGA. Special evaluation boards are developed for these purposes, one of them is presented in [9] or [10]. The systems implemented as fault-tolerant very often consist of two parts - an electronic one and a mechanical one. The mechanical part is controlled by its electronic controller. It can be stated that such areas exist in which electro-mechanical applications are implemented as fault-tolerant - aerospace and space applications can serve as an example. The platforms for the verification of fault-tolerant qualities that allow us to just chceck the resilience of the electronic component have been used until now. *We feel that for electro-mechanical systems the approach must be different. It must be possible to check what are the reactions of the mechanical component if the functionality of its electronic controller is corrupted by external attacks.*

The basic concepts and the first version of evaluation platform were presented in our previous work [11]. The first version of the evaluation platform is composed of three parts:

978-1-5090-2817-7/16 $31.00 © 2016 IEEE
DOI 10.1109/DSD.2016.38

1) robot controller running on FPGA, 2) simulation of the robot and its environment running on PC and 3) previously developed fault injector [12] running on PC. Based on experiments with our platform we realized the necessity to automate the process of a fault impact evaluation. We found functional verification as an appropriate technique for this purpose.

The proposed process of the fault impact evaluation, which is shown in Figure 1, is divided into three phases. In the first phase, we use the simulation-based functional verification where the VHDL description of the electronic robot controller is used as the DUT. In this phase, the correctness of the robot controller is evaluated. The second phase consists of the verification of the robot controller implemented into FPGA with the scenarios obtained during the previous phase and uses a previously implemented fault injector. The analysis of the faults which corrupted the mechanical part is the goal of the third phase. The development of the verification environment and the development of a reference model for the electronic control unit (the robot controller) are the first steps towards this process. Both of these activities are described in detail in this paper. The second step is to implement DUT to FPGA and its interconnection with the simulation environment of robot. The architecture of the verification environment with the robot controller implemented to FPGA is also presented in this paper. The experiments which correspond with the first and the second phases of the proposed process are also important parts of our work.

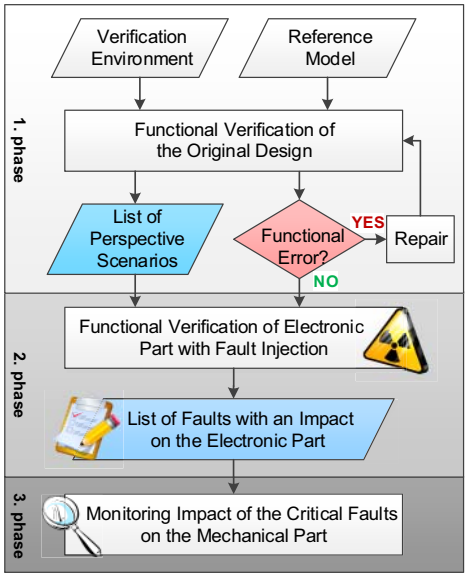


Fig. 1. The flow of phases in the digital systems verification.

The main output of the first phase is a test on whether the robot controller works correctly according to the specification. It is important because we have to ensure that the robot controller does not contain any functional errors in the implementation. It is also important to point out that in this phase we acquire a set of verification scenarios (different mazes with different start and goal positions for robot movements) that will also be used in the subsequent phase. One verification run is represented by the robot moving through the maze from the start position to the goal position.

The outputs of the second phase are previously verified verification scenarios supplemented by information about injected faults and its impact on the electronic part. The injected faults are divided into two categories, faults with no impact on electronic part and faults which cause mismatches on the output of the electronic part. Various strategies of fault injection may be used in this phase (e.g. one fault for one verification run, multiple faults in the same functional unit or multiple faults in different functional units).

This paper is organized as follows. The architecture of the verification environment for the first phase is described in Section II. Section III describes evaluation platform architecture used in the second phase. The principles of generating verification scenarios are described in Section IV. Section V shows experiments and results corresponding with the first and second phases of the evaluation process. Section VI summarizes the results and proposes our plans for future research.

II. The First Phase - Verification Environment Architecture

The verification environment architecture, its basic components and used techniques are described in this section. First, UVM based verification environment for one verification scenario (one maze, start and goal positions) is presented, which forms the core of an extended verification environment for multiple verification scenarios evaluation.

A. Verification Environment for Single Verification Scenario

The verification environment for the robot controller is designed according to UVM, so it corresponds with current trends and requirements. The basic architecture of the verification environment with main components is shown in Figure 2 [13]. It should be noted that the verification environment is connected with the robot in the maze (the robot in the maze is simulated in simulation environment Player/Stage [14]). The robot in the maze is controlled by the outputs of the robot controller (DUT) while the outputs of the robot in the maze (information from sensors) are inputs to the robot controller. The information whether DUT satisfies (or does not satisfy) specification and coverage report for the verified scenario are the outputs of the verification environment. These are the components of the system together with their description:

- *The robot controller* under verification implemented in VHDL is able to search a path through a maze. Detailed information is available in [15].
- *The golden (reference) model* implemented in C/C++ according to the same specification as the robot controller performs the same operations as DUT. The reference model is described in detail in [13].
- *The sequence* is the component which receives data from sensors placed in the robot in the maze. Received data (information about barriers in four neighborhoods and the position in the maze) are transformed to the inputs of the verification environment.
- *The driver* sends input values (data from sensors) to reference model and DUT (robot controller).
- *The monitor* reads the outputs from DUT (speed of the robot in the maze) and forwards them to the scoreboard and to the robot in the maze which moves according to these values.

- *The scoreboard* compares the outputs of the monitor and reference model on equality and checks mismatches on the outputs. The detected mismatch shows that there are differences between DUT and reference model outputs.

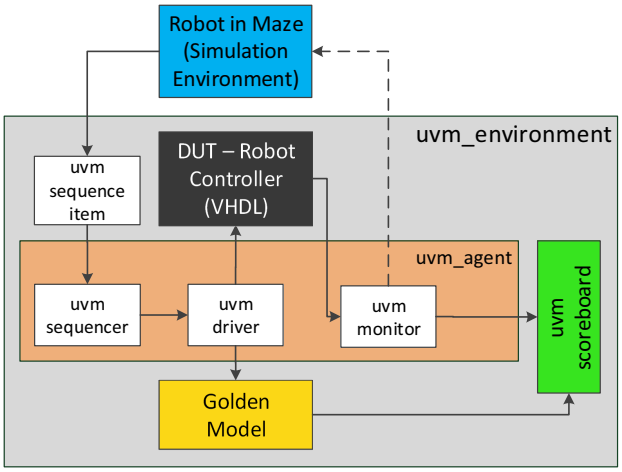


Fig. 2. Verification environment for single verification scenario.

B. Extended Multiple Verification Scenarios Evaluation

The presented verification environment is not able to evaluate multiple verification scenarios automatically and we need the extension of the process to be automated. The extension of the verification environment is presented in this section. The verification environment is used as one of several components. Other components such as maze random generator are also important. The design of the complete extension is shown in Figure 3. The components, their inputs, outputs and connections are shown in the figure and their description is as follows:

- *The maze generator* allows us to generate a sufficient number of mazes with respect to specified parameters (size, width of corridor etc.) in order to achieve the required coverage. In our work, we use a maze generator based on our universal generating principle described in Section IV.
- *The robot simulation* replaces the real robot because we do not have a real one. As mentioned above, we use the Player/Stage [14] simulation environment which provides features that we need for our research.
- *The step counter* calculates the number of steps that the robot must perform to pass from the starting to the goal position. This information is important for proper operation of the UVM verification environment.
- *The UVM verification environment* is the core of the extended evaluation.
- *The verification scenario* allows us to use it in the second phase which uses a fault injector (Figure 1). A certain part of the stored verification scenario is also a report about the coverage which was obtained by this scenario.
- *Merge the coverage* achieved by the single verification scenario is important to obtain a final coverage report gained by stored sets of verification scenarios.

Figure 3 also shows the outputs of the first phase of the fault impact evaluation process presented in Section I which are *Set of Verification Scenarios* and obtained *Total Coverage Report*.

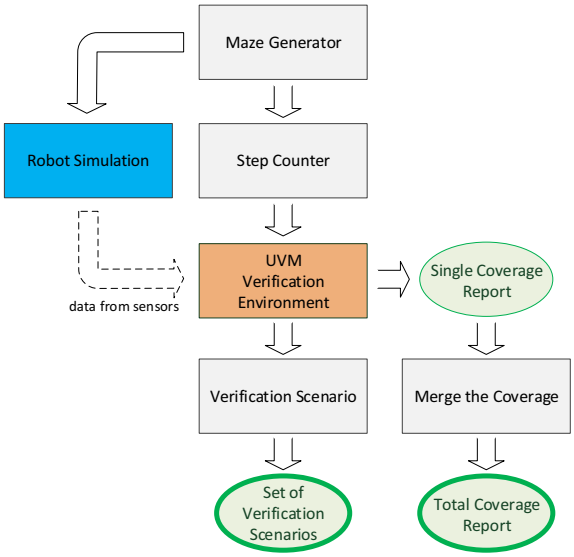


Fig. 3. Extension of verification environment for multiple evaluation.

III. The Second Phase - Evaluation Platform Architecture

The second phase of the evaluation process is functional verification of the design implemented to the FPGA. Moreover, the fault injection into the FPGA is performed in this phase. The experimental platform was designed for these purposes which is composed of a few components running on a computer or on an FPGA evaluation board:

1) software part of verification environment for the robot controller running on computer,
2) software simulation environment for robot simulation (Player/Stage) running on computer,
3) robot controller implemented to FPGA, and
4) external fault injector [12] running on a computer which allows us to simulate real faults in FPGA.

The overall experimental platform interconnection is shown in Figure 4. The connection between a computer and an FPGA is realized by JTAG and Ethernet. JTAG interface is used for FPGA programming and the software and hardware part of verification environment are connected through Ethernet. The fault injector also uses JTAG for placing faults into the FPGA configuration memory. The description of the architecture of the verification environment and of the fault injection process follows.

A. Architecture of FPGA-based verification environment

For these purposes, the FPGA-based verification environment which is displayed in Figure 5, is derived from the version created in the first phase. The architecture of the verification environment is divided into two parts. The first part is the simulation environment of a robot in the maze

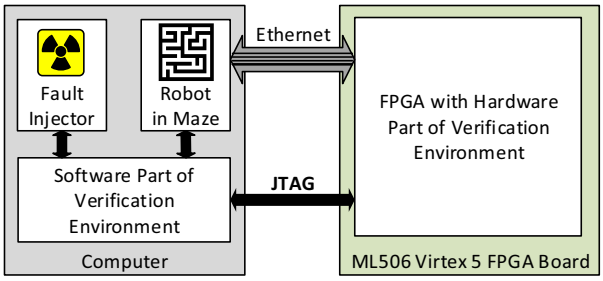


Fig. 4. The structure of the experimental platform.

which is controlled by the robot controller implemented to FPGA. The communication between the software and the hardware part is accomplished using a proprietary interface (more details about the communication are provided in the subsequent subsections). This part operates autonomously, the robot controller receives information from the robot sensors which are produced by the simulation environment and sends them to the FPGA through Output Wrapper. On the other hand, speed and direction of movement are sent through Input Wrapper from the robot controller implemented in FPGA to the robot in a simulation.

The second part is the UVM-based verification environment which operates as an observer without direct intervention to data transfers between the robot controller and robot in a simulation environment. The verification environment just checks the correctness of transferred data which are resent to the verification environment as can be seen in Figure 5. Information from sensors is received in the Sequence component where they are transformed to transactions and transferred to the Golden Model which produces reference output data. Speed and direction of movement are received in the Monitor component and sent to the Scoreboard component. The Scoreboard compares received data with reference data obtained from Golden Model.

Both parts are synchronized by signals sent from the Sequence and Monitor components to the robot simulation environment. These signals indicate that the verification environment is ready to observe robot movement in the maze.

Presented FPGA-based verification environment evaluates only one verification scenario, but automated evaluation of multiple verification scenarios with fault injection is needed. The second phase eliminates the need for maze generation because mazes pregenerated and verified in the first phase are used. Conversely, there are new steps as a consequence of implementing robot controller into FPGA and the creation of an autonomous connection between the FPGA and robot in the maze. The first necessary step is programming the FPGA through JTAG interface which must be done before each verification run. This step ensures that the correct functionality of the robot controller is verified and is without faults. Programming FPGA clears BRAM memory where a map of the maze is continuously stored which is important when the maze is changed.

The next step is launching the robot in a simulation and verification environment which provides enable signals to the simulation environment when it is ready to start monitoring. Then, the robot starts to search for paths through the maze

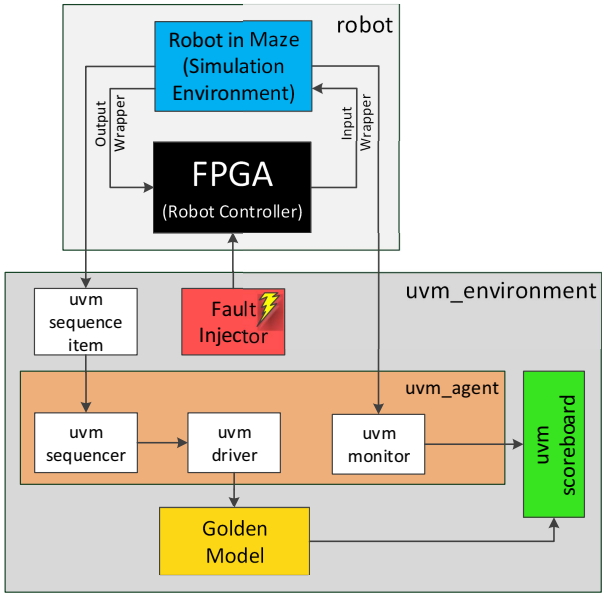


Fig. 5. The architecture of the FPGA-based verification environment.

which is the proper time for fault injection. It should be noted that fault injection proceeds according to the selected strategy. Our fault injector allows us to inject faults into specified functional units which can be advantageously used. For example, we can inject single faults during one verification run into the specified functional unit, multiple faults into the specified functional unit or inject multiple faults into multiple functional units. After fault injection, the verification run is finished or timeout is expired and then results of the verification are recorded into the verification report.

B. Communication Between Software and Hardware Part

Communication between the robot controller implemented on the FPGA (hardware part) and robot in a simulation environment (software part) is accomplished through Input and Output Wrapper. We chose ML506 development board [16] with Xillinx Virtex 5 FPGA as the hardware platform. This board offers various peripherals and some of them can provide communication with a PC (e.g. PCIe, UART, USB or Ethernet). We decided to use Ethernet communication because of its versatility. The chip implementing the Ethernet physical layer is connected to the FPGA and user design which implements higher layers of the Ethernet protocol stack that can communicate with this chip. However, we do not implement a full Ethernet protocol stack, but use an existing implementation presented in [17].

Figure 6 shows the architecture of the communication layer. Although we use an existing implementation of Ethernet communication we must solve a problem with different clock signals on receive (RX) and transmit (TX) interfaces. These clock signals are generated by a physical layer chip and the designer is not able to modify the frequency and phase offset. We use FIFO memory as an input and output buffer with different writing and reading clock signals. This solves not only the problem with clock domain crossing, but also the problem with data storing before their processing. Received

data from Ethernet are buffered in the input buffer and data ready to be sent are buffered in the output buffer. We use FIFO as the interface of DUT which allow us to exchange a communication layer with another one which uses FIFO buffers.

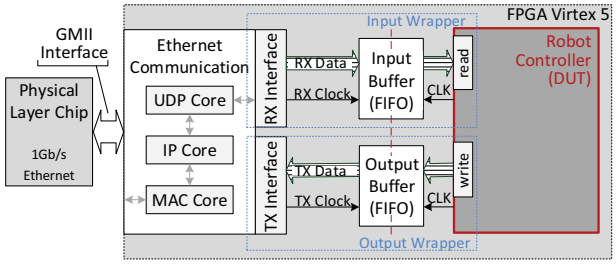


Fig. 6. The architecture of communication between SW and HW part.

C. Evaluation of Reliability by Fault Injection

The simulation of the effects of faults in the FPGA can be done by a direct change of the configuration bitstream which is loaded into the configuration memory. For this purpose, we implemented a fault injector [12] which allows us to prepare the bitstream for our FPGA and also modify single or multiple bits of the bitstream in order to simulate single and multiple faults. As a consequence, the design placed in the FPGA (determined by the configuration data) is similarly influenced by a real fault which strikes the hardware architecture of the FPGA in a real environment.

The injector is based on the SEU generation outside of the FPGA (in PC), so it is not targeted to a specific FPGA board (testing was performed on the ML506 card with the Virtex 5 FPGA technology). The original and the modified bitstream is transported through the JTAG interface. The process of the SEU generation is divided into four steps: 1) specifying the location of the fault injection, 2) reading the related part of the configuration bitstream, 3) the SEU generation (i.e. the inversion of the specified bit of the bitstream), and 4) applying the bitstream using *Partial Dynamic Reconfiguration* (PDR) without stopping the FPGA.

The implemented fault injector is able to inject a fault into a specified bit of bitstream. If we are able to find a relation between bits of bitstream and functional units, we can inject faults into the specified functional unit. For this purpose, the analysis of FPGA can be done by RapidSmith [18] tool. This tool identifies the bits of bitstream which are related with a specified area on the FPGA. Functional units placement on the FPGA is done by PlanAhead [19] tool, then we know where each of the functional units are placed. This process allows us to inject faults into specified functional units during our experiments. Unfortunately, the process actually finds only the bits of the bitstream corresponding with Look-up tables (LUTs).

IV. Maze Generation

Maze generation is a well known and explored area for which a considerable number of algorithms generate simple or sophisticated mazes that exist [20]. The vast majority of algorithms operate in a two-dimensional space, keep the current state and can constantly change cell values of a maze in time. These algorithms are highly unsuitable for our proposed architecture of the universal generator [21], because the output of the generator (a line of the maze) cannot be determined in one step, therefore, it is determined gradually by many factors and dependencies between different cells of the maze. However, an algorithm exists that is based on a binary tree and a particular line of the maze can be determined only from the previous one. This principle is completely satisfactory for our generator and the output maze is fully sufficient for our needs.

The basic principle of the binary tree algorithm is shown in Figure 7. It starts from the basic matrix of the maze (a) in which some cells are tightly specified - either the corner or the wall. We represent the corridors by zeros and the walls by ones. Cells marked with a question mark represent areas that can take the value of 0 or 1, thus the corridor or the wall. Inorder to maintain the continuity from any corner of the maze to another, it is necessary to perform modification of the basic matrix of the maze so that each two adjacent sides of the maze must contain the corridor over its entire dimension (b). In our case, we chose this corridor to the northern and the eastern side of the maze. The final most critical task is to determine the cells A, B, C, D which allows us to have the maximal continuous maze (c).

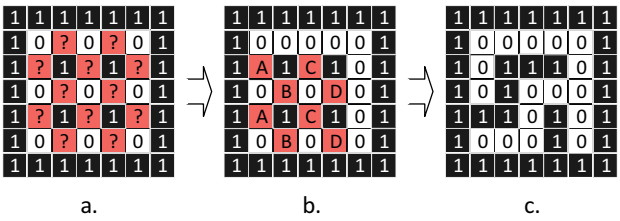


Fig. 7. The demonstration of a conversion of the basic matrix of the maze for needs of the generator.

The original description of the algorithm [20] divides cells of the maze in a line into groups of corridors bordered by walls. For each group, an algorithm determines one entrance, either in the northern or in the eastern part of the border. This ensures the passage from the northern part of the maze to the south and the same applies for the passage from the west to the east. We transferred this principle into one line dependency in the maze and the result is the following dependence. If a cell A respectively C was randomly selected for the corner in Figure 7.b, then the cell B respectively D will be a wall and vice versa.

The architecture of the universal generator is based on two input structures - the Problem Description and Constraints. In this case, the Problem Description defines a set of values and desired output format - zeros and ones. Constraints represent restrictions based on the preceding paragraph which are required for the continuous generation of the maze. The samples of simplicity of both structures, without further explanation, are available below. The structures are sufficient to generate the maze with 7x7 cells.

```
----- Problem Description -----

substitute {
    A,C { "0"|"1" }
    B,D { "0" }
}
```

```
syntax {
    odd { "1A1C101" }
    even { "10B0D01" }
}

----- Constraints -----

constraints {
    nlines(7,7)

    ifthen(A("0"),B("1"))
    ifthen(C("0"),D("1"))

    start("1111111")
    start("1000001")

    useonly(odd)
    afterinsert(odd,even)

    end("1111111")
}
```

We continued in our previous research published in [11] by this maze generation and we have shown another possible area for our architecture of the generator. Any desired dimension of the maze can be generated with minor modifications. In order to use an assumption of the basic matrix of the maze, it is necessary to choose the odd dimensions of mazes. In our previous work, we were able to generate assembler programs for RISC and VLIW processors [21] which is a completely different type of input stimuli for the same generator.

V. Experiments and Results

Performed experiments correspond to the activities of the first and second phase of the fault impact evaluation process.

A. The First Phase

The outputs of the first phase are: 1) the electronic part without bugs (robot controller), 2) the list of the used verification scenarios, and 3) achieved coverage. Figure 8 shows three types of mazes which we used in our experiments. The presented mazes differ in their dimensions, we chose 7x7, 15x15 and 31x31 cells. Examples of start and goal positions are also shown in Figure 8. With the growing size of the maze the number of steps that the robot must also go through increases. The average number of the robot steps in various types of mazes is shown in Table I. The main goal of the experiment, including debugging the robot controller, was to determine the optimal size of the maze and the number of generated mazes (verification scenarios) which will lead to the best code coverage.

TABLE I. Average number of robot steps

Maze size	7x7	15x15	31x31
Average number of steps	16	93	433

For the experiment, we chose the number of performed verification scenarios equal to 10, 100, 200 and 500, for which we monitored achieved code coverage. The numbers of performed verification scenarios were the same for all types of mazes, in total 1,500 verification scenarios were performed with a variety of mazes. Various bugs were identified and

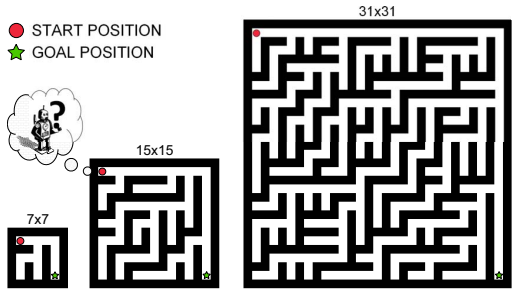


Fig. 8. Three types of mazes.

debugged during the verification process. We can state that the robot controller operates according to its specification for the performed 1,500 verification scenarios.

Experimental results are presented in Table II. It can be recognized that the maximal achieved total code coverage is 91.85%. The missing percentage to an ideal 100% is caused by the "others" branches in the source code which are never executed (which is correct), and also by some of the control expressions that are used only when an abnormal situation occurs (e.g. fault). The table also shows that a rising number of verification scenarios does not increase the achieved code coverage. It is probably because in one scenario multiple input transactions are packed.

On the other hand, resizing the maze from 7x7 to 15x15 cells led to a slight increase of code coverage, suggesting the effect of the maze. When increasing the size of maze to 31x31 cells, the coverage was not changed. Such studies show that the 7x7 cells maze is too small for the next phase of fault impact evaluation process. This trend is shown in a bar chart in Figure 9 which shows the code coverage for different sizes of mazes for 100 verification scenarios.

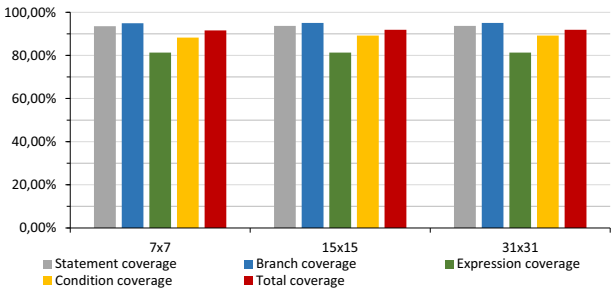


Fig. 9. Code coverage for each type of mazes for 100 verification scenarios.

The results needed to perform the next phase of the fault impact evaluation were obtained in the experiment. Faults will be injected into the electronic controller during each verification scenario in the second phase of evaluation. Each verification scenario will be repeated several times and during each run various faults or various sequences of faults will be injected.

B. The Second Phase

The second phase in the proposed evaluation process is targeted towards evaluating the correct function of robot controller implemented into the FPGA. For this purpose fault injection is used. No fault tolerance methodology implemented

TABLE II. THE EXPERIMENTAL RESULTS

# of verification scenarios	10			100			200			500		
Size of mazes	7x7	15x15	31x31	7x7	15x15	31x31	7x7	15x15	31x31	7x7	15x15	31x31
Statement coverage	93,54 %	93,70 %	93,70 %	93,54 %	93,70 %	93,70 %	93,54 %	93,70 %	93,70 %	93,54 %	93,70 %	93,70 %
Branch coverage	94,91 %	95,07 %	95,07 %	94,91 %	95,07 %	95,07 %	94,91 %	95,07 %	95,07 %	94,91 %	95,07 %	95,07 %
Expression coverage	81,33 %	81,33 %	81,33 %	81,33 %	81,33 %	81,33 %	81,33 %	81,33 %	81,33 %	81,33 %	81,33 %	81,33 %
Condition coverage	88,28 %	89,18 %	89,18 %	88,28 %	89,18 %	89,18 %	88,28 %	89,18 %	89,18 %	88,28 %	89,18 %	89,18 %
Total coverage	91,61 %	91,85 %	91,85 %	91,61 %	91,85 %	91,85 %	91,61 %	91,85 %	91,85 %	91,61 %	91,85 %	91,85 %

in the robot controller for these experiments was used and the goals of the experiment are: 1) detailed reliability analysis of the robot controller and its functional units, and 2) demonstration that the evaluation platform can be used for fault tolerance evaluation.

Before explaining the details of our experiments, we must introduce the robot controller which consists of various blocks, whose function is described in [15]. The controller is connected to the PC on which robot simulation environment (SEPC) runs via the Interface Block. Through this block, data from the simulation are received and in the opposite direction, instructions defining the required movement of the robot are sent back. The central block of the robot controller is a bus through which the communication between blocks is accomplished. The Position Evaluation Unit (PEU) calculates the positions of the robot in the maze and provides them to other units as coordinates x and y. The Barrier Detection Unit (BDU) uses four sensors and provides information about the distance to the surrounding barriers. The map updating provided by the Map Unit (MU) is based on information about the positions of the robot and the barriers vector. The Map Memory Unit (MMU) stores the information about an up-to-date map. Path Finding Unit (PFU) implements a simple iteration algorithm for finding a path through the maze. The mechanical parts of the robot are driven by setting the speed in the required direction of the movement by the Engine Control Module (ECM). The communication of functional units with bus is accomplished through the bus wrapper (FU_WB) and controlled by the finite state machine (FU_FSM).

As was mentioned above, faults can be injected in a way which reflects various strategies. Similar experiments were done in our previous work [11] but significant differences in evaluation strategies are presented in this paper. We have decided to perform 50 verification runs and inject one fault into one functional unit (single fault) during one verification run and to use the mazes of larger dimensions, the mazes of 15x15, for this phase. The robot controller consists of 15 functional units which leads to 750 verification runs and injected faults in total. The task of the verification environment was to compare the outputs of the robot controller and check the impact of injected fault. Table III shows the number of verification runs where the incorrect outputs of the robot controller were caused by faults (percentage values are shown as well). The total number of verification runs for each functional unit is 50 and the main reason for this is the time complexity of the verification runs, because the robot has to go through the whole maze.

The results of our experiments are shown in Figure 10 as well. The bar chart expresses a percentage number of faults with their impact on the robot controller. Horizontal lines in the chart show minimum, average and maximum values. As can be seen, some anomalies in the results of the experiments

TABLE III. EXPERIMENTAL RESULTS IN FUNCTIONAL VERIFICATION.

Unit	Number of fails	Fails in %	Unit	Number of fails	Fails in %
bdu	40	80	mu_wb	31	2
bdu_fsm	19	38	peu	39	78
bdu_wb	35	70	peu_fsm	40	80
ecu	38	76	pfu	34	68
intercon	31	62	pfu_wb	28	56
mmu	31	62	sif_fsm	50	100
mu	25	50	sif_wb	34	68
mu_fsm	1	2			

exist. These include results combined with three functional units *mu_fsm*, *peu_fsm* and *sif_fsm*. In the case of *mu_fsm*, it is apparently a low number of faults with an impact on the correct function of the robot controller. Functional units *peu_fsm* and *sif_fsm* represent a completely different situation, the number of faults with impact is significantly higher than for other units. That is why we repeated the experiments on a higher number of verification runs (225 in this case) with these functional units. Table IV shows additional verification runs that was performed in order to analyse these anomalies in detail. As can be seen, the additional results are closer to the overall average.

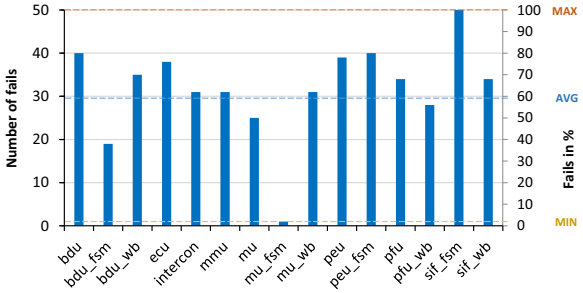


Fig. 10. Experimental results in functional verification.

We have made the fault injection analysis of the robot controller. We have found out that some blocks are more prone to faults than others. As can be recognized in the chart showing the results, the functional unit *mu_fsm* is less prone to faults than the other units. On the other hand, the units *peu_fsm* and *sif_fsm* are the most prone units to faults. This is especially important for future application of fault-tolerant methodologies. A system designer obtains the information which blocks needs more attention from a reliability point of view.

The second finding is that we are able to use the functional verification in conjunction with the fault injector to determine the impact of faults on the electro-mechanical system. Our system could be used to automate the evaluation of fault tolerance methodologies after these methodologies are applied to the electro-mechanical system (in our case the robot controller).

VI. CONCLUSIONS AND FUTURE RESEARCH

In this work, we introduced a verification environment which shows the progress of our research. The verification

TABLE IV. EXTENDED EXPERIMENTAL RESULTS.

Unit	Number of fails	Fails in %
mu_fsm	18	8
peu_fsm	181	80.4
sif_fsm	219	97.3

environment is the main part of our platform for evaluating fault impact on the electro-mechanical system. The introduced basic verification environment is able to evaluate a single verification scenario and the creation of an extension which allows automated evaluation of multiple verification scenarios was presented as well. This automated evaluation uses the maze generator based on our universal generator approach. The verification environment for the second phase where the DUT is implemented to the FPGA was also created. In the proposed methodology, the verification environment acts as an observer that checks data transferred between the electronic and mechanical part.

Performed experiments correspond to the first and second phases of a fault impact evaluation process. The output of the first phase is the debugged electronic controller and the list of verification scenarios for the next phase. During the second phase, the reliability analysis was done by means of the fault injection into the FPGA. The result is the ratio of faults that caused an incorrect output of the electronic controller.

In our future research, we shall prepare experiments corresponding with the third phase of the proposed evaluation process which checks reactions of the mechanical part, not only of the electronic part. We must create the extension of our evaluation platform for these purposes. Thanks to the Player/Stage simulation environment we are able to receive not only information from sensors, but also information about the behaviour of a robot in the maze. Next, the goal of our future work is to apply various fault tolerance methodologies on the robot controller and evaluate them with our evaluation platform. For example, we plan to construct our robot controller as a fault tolerant neural network. We can use more conventional fault tolerant methodologies such as TMR, online checkers or error correction codes. We will focus on testing fault tolerance methodologies targeted to FPGAs in the context of electro-mechanical systems which is often the way of using fault-tolerant electronic controllers. On the basis of these results, we are going to develop generally usable principles of developing systems for evaluating fault tolerant qualities of electro-mechanical systems.

ACKNOWLEDGMENT

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 641439 (ALMARVI) and BUT project FIT-S-14-2297.

REFERENCES

[1] A. Benso and P. Prinetto, *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, ser. Frontiers in Electronic Testing. Springer Science & Business Media, 2003, vol. 23.

[2] A. Meyer, *Principles of Functional Verification*. Elsevier Science, 2003. [Online]. Available: http://books.google.cz/books?id=qaIiX3hYWL4C

[3] V. R. Cooper, *Getting Started with UVM: A Beginner's Guide*. Austin, TX : Verilab, 2013.

[4] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[5] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam, "Mitigation of Radiation Effects in SRAM-Based FPGAs for Space Applications," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 37:1–37:34, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2671181

[6] XILINX. (2014, Nov.) FPGA. [Online]. Available: http://www.xilinx.com/fpga/index.htm

[7] M. Ceschia, M. Violante, M. Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, and A. Candelori, "Identification and Classification of Single-event Upsets in the Configuration Memory of SRAM-based FPGAs," vol. 50, no. 6, 2003, pp. 2088–2094.

[8] C. Bernardeschi, L. Cassano, A. Domenici, and L. Sterpone, "Accurate Simulation of SEUs in the Configuration Memory of SRAM-based FPGAs," in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 115–120.

[9] M. Alderighi, S. D'Angelo, M. Mancini, and G. R. Sechi, "A Fault Injection Tool for SRAM-based FPGAs," in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*. IEEE, 2003, pp. 129–133.

[10] M. Alderighi, F. Casini, S. d'Angelo, M. Mancini, S. Pastore, and G. R. Sechi, "Evaluation of Single Event Upset Mitigation Schemes for SRAM-based FPGAs Using the FLIPPER Fault Injection Platform," in *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT'07. 22nd IEEE International Symposium on*. IEEE, 2007, pp. 105–113.

[11] 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*. IEEE, 2014, pp. 312–319.

[12] M. Straka, J. Kastil, and Z. Kotasek, "SEU Simulation Framework for Xilinx FPGA: First Step Towards Testing Fault Tolerant Systems," in *14th EUROMICRO Conference on Digital System Design*. IEEE Computer Society, 2011, pp. 223–230.

[13] S. Krajcir, "Functional Verification of Robotic System Using UVM," Tech. Rep., 2015. [Online]. Available: http://www/study/DP/DP.php?id=15154

[14] B. Gerkey, R. T. Vaughan, and A. Howard, "The Player/Stage Project: Tools for Multi-robot and Distributed Sensor Systems," in *Proceedings of the 11th international conference on advanced robotics*, vol. 1, 2003, pp. 317–323.

[15] J. Podivinsky, M. Simkova, and Z. Kotasek, "Complex Control System for Testing Fault-Tolerance Methodologies," in *Proceedings of The Third Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale (MEDIAN 2014)*. COST, European Cooperation in Science and Technology, 2014, pp. 24–27.

[16] Xilinx Inc., "Ml506 Evaluation Platform User Guide," *UG347 (v3. 1.2)*, 2011.

[17] P. Skibik, "Implementation of Ethernet Communication Interface into FPGA Chip," Tech. Rep., 2011. [Online]. Available: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=40494

[18] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings, "Rapid Prototyping Tools for FPGA Designs: RapidSmith," in *Field-Programmable Technology (FPT), 2010 International Conference on*, Dec 2010, pp. 353–356.

[19] N. Dorairaj, E. Shiflet, and M. Goosman, "PlanAhead Software as a Platform for Partial Reconfiguration," vol. 55, no. 84, 2005, pp. 68–71.

[20] Jamis Buck. (2011, Feb.) Maze generation: Algorithm recap. [Online]. Available: http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap

[21] J. Podivinsky, O. Cekan, M. Simková, and Z. Kotásek, "The evaluation platform for testing fault-tolerance methodologies in electro-mechanical applications," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 39, no. 8, pp. 1215–1230, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2015.05.011

Paper IV

A Probabilistic Context-Free Grammar Based Random Test Program Generation

ČEKAN Ondřej, KOTÁSEK Zdeněk
In: *Proceedings of 20th Euromicro Conference on Digital System Design.* Vídeň: Technische Universität Wien, 2017, pp. 356-359. ISBN 978-1-5386-2146-2.

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).

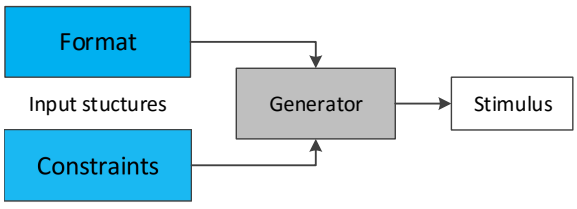


Fig. 1: The architecture of the universal stimuli generation.

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∩T = 0.
- R — is a finite set of rewrite rules with form A $\rightarrow \alpha$, where A ∈ N a α ∈ (N∪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 ∈ 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.

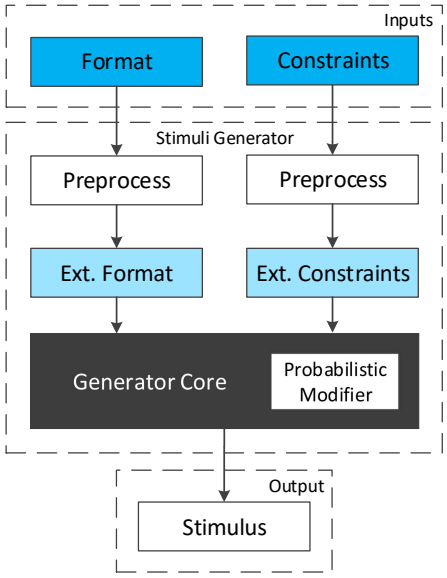


Fig. 2: The detailed architecture for a probabilistic context-free grammar based stimuli generation.

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

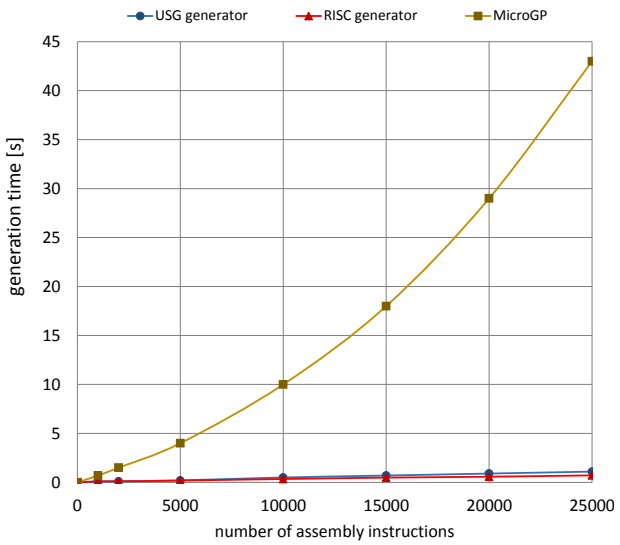


Fig. 3: The comparison of generation time for USG, RISC, and MicroGP tool.

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.

Paper V

Program Generation Through a Probabilistic Constrained Grammar

ČEKAN Ondřej, PODIVÍNSKÝ Jakub, KOTÁSEK Zdeněk
In: *Proceedings of the 2018 21st Euromicro Conference on Digital System Design*. Praha: IEEE Computer Society, 2018, pp. 214-220. ISBN 978-1-5386-7376-8.

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

- 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.

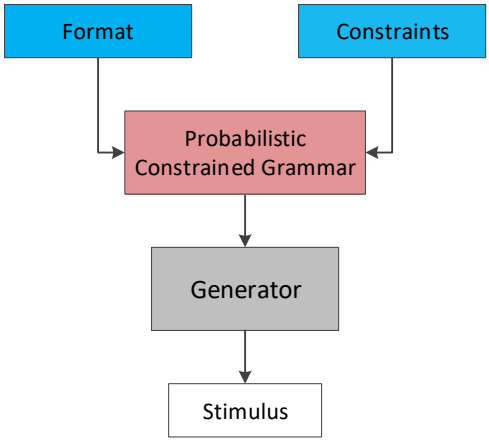


Fig. 1: The versatile architecture of stimuli generation.

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.

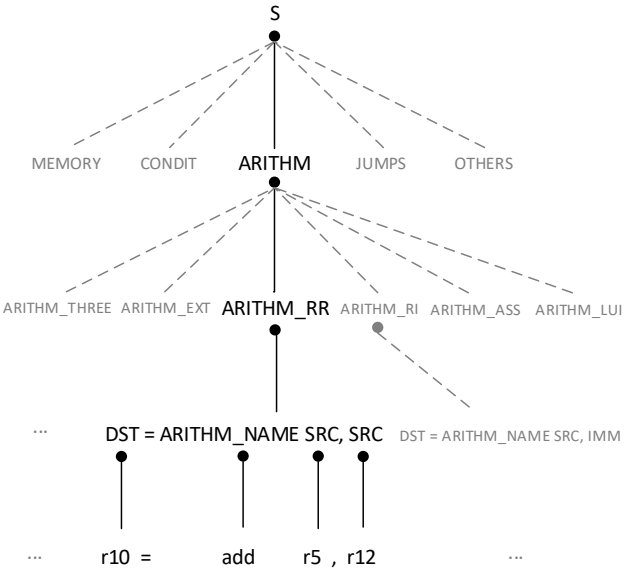


Fig. 2: The derivation tree for ADD instruction which is composed of destination register, instruction name, and two source registers.

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 %}
```

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.

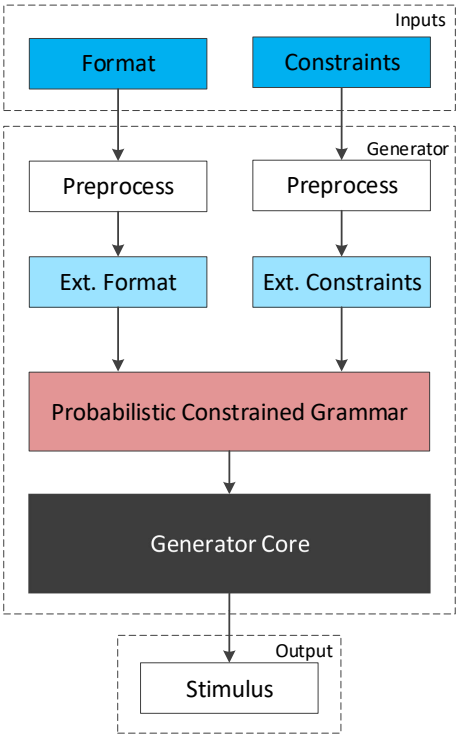


Fig. 3: The detailed architecture of the stimuli generation based on the grammar system.

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
```

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%

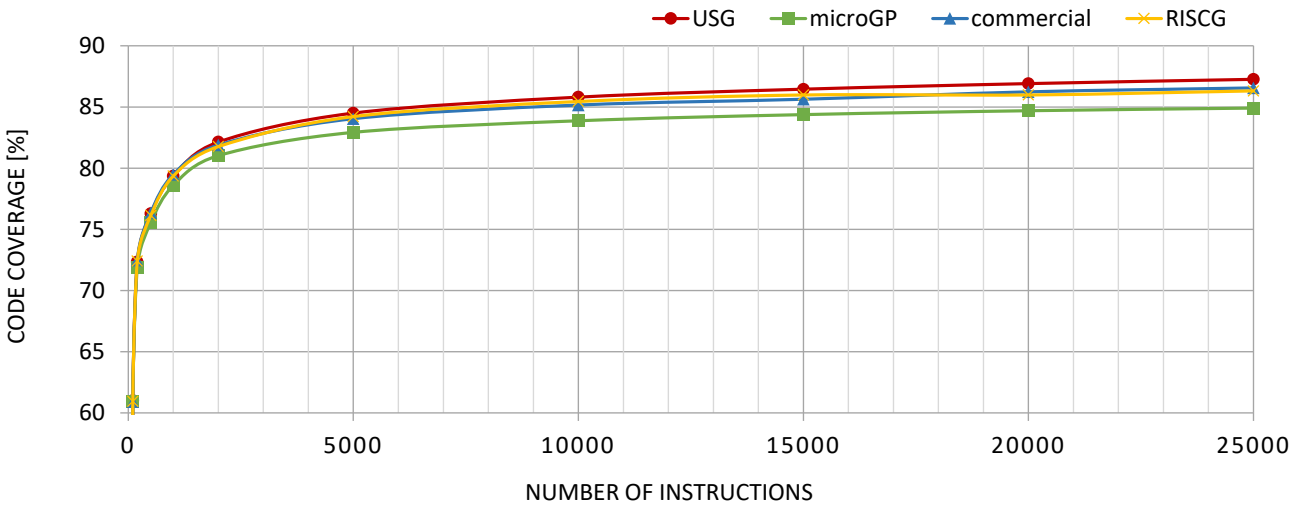


Fig. 4: The comparison of achieved code coverages in the functional verification for 100 programs.

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.

Paper VI

Input and Output Generation for the Verification of ALU: A Use Case

ČEKAN Ondřej, PÁNEK Richard, KOTÁSEK Zdeněk
In: *Proceedings of IEEE East-West Design & Test Symposium.* Kazan: IEEE Computer Society, 2018, pp. 331-336. ISBN 978-1-5386-5709-6.

Input and Output Generation for the Verification of ALU: a Use Case

Ondrej Cekan, Richard Panek, Zdenek Kotasek
Brno University of Technology, Faculty of Information Technology,
Centre of Excellence IT4Innovations
Bozetechova 2, 612 66 Brno, Czech Republic
Tel.: +420 54114-{1361, 1362, 1223}
{icekan, ipanek, kotasek}@fit.vutbr.cz

Abstract

The paper presents the approach to universal stimuli generation for an arithmetic-logic unit (ALU). It is not focused only on input data generation, but it is possible to generate also expected output in one stimulus. The process of generation is based on a probabilistic constrained grammar which is designed to universally describe stimuli for various circuits. This grammar is processed by our framework. The experiment in functional verification, which shows the quality of generated stimuli, is also presented.

1. Introduction

Random stimuli generation is currently a very important process of checking the correct behavior of various circuits [1]. Complex or also simple circuits must be properly tested or verified before real deployment to exclude design or implementation errors. It is also necessary to verify the correct output for expected and unexpected input combinations (stimuli). Stimuli are typically randomly constructed and may take many forms from binary values on simple circuit pins to a complex program in the data memory of a processor.

Each system is unique, and therefore, it requires specific input stimuli for its operation. In order to verify the correct behavior, it is necessary to create a set of test cases (input stimuli and expected outputs) to detect any possible mismatches in the circuit. Depending on the complexity of the circuit, this activity may be quite challenging, and therefore, tools that allow to generate random inputs automatically are created. These tools are targeted to a specific circuit and their use is considerably limited for different devices. Also, these tools do not allow the expected output to be generated, and further efforts must be made to create a reference system [2].

For the reasons outlined above, we have focused on developing a framework for universal stimuli generation that can be used for various circuits.

The paper is organized as follows. In section 2 our previous research is described. In section 3 the related work is summarized. Section 4 deals with our definition of probabilistic constrained grammar that we use for the generation process while section 5 devotes to the grammar definition for the arithmetic-logic unit. Experimental results are mentioned in section 6 and finally in section 7 the paper is concluded.

2. Previous Research

In our previous research, we designed and developed a framework for universal stimuli generation based on a probabilistic (stochastic) context-free grammar [3]. It is a common context-free grammar that defines probabilities for its production rules with which they are applied. We have extended this grammar by restrictive conditions (constraints) and defined the new grammar system - Probabilistic Constrained Grammar (PCG) [4] that we use in our research. Constraints are used to dynamically change the probabilities of production rules during the generation.

We have also defined the architecture of universal stimuli generation [5] that is shown in Fig. 1. This architecture consists of two input structures (Production Rules, Constraints) which are based on PCG. The first structure defines the production rules of a grammar, while the second structure includes constraints for the application of production rules. Together these two structures form the resultant grammar. Grammar defined in this way is processed by

the Generator Core of the framework that assembles the resultant stimulus on its output.

We applied this framework to more complex circuits (e.g., RISC (Reduced Instruction Set Computer) [6] processors, control unit) to verify the possibility of generating stimuli using PCG. We verified the quality of obtained stimuli from the point of view of the generation speed and the achieved coverage [7] in functional verification.

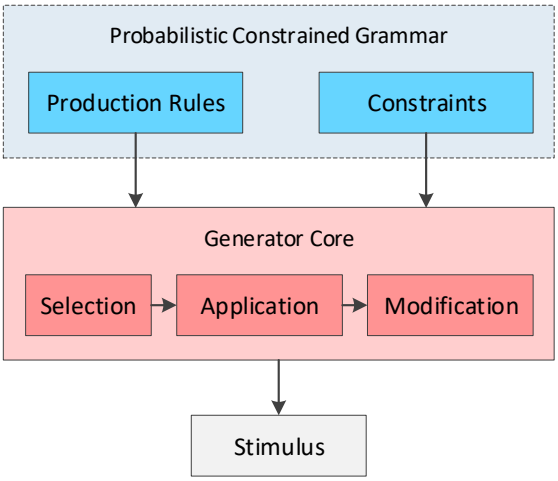


Fig. 1: The architecture of universal stimuli generation.

3. Related Work

The current trend in stimulus generation focuses primarily on more complex circuits (e.g. processors), because it is not trivial to construct a valid stimulus (working program). Simpler stimuli, including test vectors, can be generated directly in the simulation environment where verification takes place (e.g. Modelsim tool from Mentor Graphics [8]) or an external tool.

A number of specific stimuli generators exists for application-specific processors (ASICs) [9], digital signal processors (DSPs) [10], protocol interfaces, field programmable gate array (FPGA) converters [11], and more. These tools and their approaches are complex and their use is limited to the particular system.

As a universal stimuli generator, MicroGP tool [12] can be mentioned which does not only generate stimuli but it also finds the most optimal solution of hard problems.

In this paper, we use test stimuli which can be obtained directly from the verification environment from Modelsim tool for comparison with our approach.

4. Probabilistic Constrained Grammar

A probabilistic constrained grammar is a pair G:

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

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

A probabilistic context-free grammar is a 5-tuple H:

`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 $A \epsilon N$ and $\alpha \epsilon (N \cup T)^*$.
S is the starting non-terminal.
P is a finite set of probabilities for production rules.

Constraints restrict the grammar in the application of production rules. The constraint is a 5-tuple C:

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

R_S is the activation rule the application of which sets this constraint.
R_D is the target rule which probability is modified.
P is the new probability value.
R_E (optional) is the stop rule which application cancels this constraint.
O (optional) is the count of application of the rule R_E before canceling this constraint.

The constraints limit the application of production rules for a given non-terminal through probabilities which can be modified throughout the generation process, and therefore, we are able to control the resultant stimulus.

5. Arithmetic-Logic Unit

In general, this paper focuses on the principles of random stimuli generation which can be used for many simple circuits. It is not just generating input values for these circuits, as in our previous work, but we would like to show the expressive power of PCG and the ability to simultaneously generate as input values as output values that will be part of the resultant stimulus. Thanks to this, it is possible to check quickly the correctness of the output in case of circuit testing or functional verification.

The arithmetic-logic unit (ALU) [13] is our test case for which we show the random generation of input stimuli and their result for the selected operation. An arithmetic logic unit performs arithmetic and bitwise operations on integer binary numbers. The symbolic representation of ALU is shown in Fig. 2.

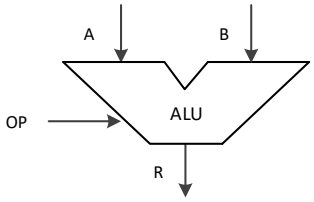


Fig. 2: The symbolic representation of ALU.

ALU has typically two input operands A and B which are N bits long. Its operation is selected by OP input bits. The R output represents the result of the operation over the operands. ALU can be variously complex, therefore, it can contain more input and output bits (e.g. status and control bits), and its supported operations can be also different in various versions.

In this paper, we limit only to inputs and outputs as shown in the figure. Among the operations under consideration, we include two arithmetic operations – addition with carry (ADD) and subtraction (SUB), and four bitwise operations - AND, OR, XOR and NOT. However, the principles that we use for the generation are applicable to other operations.

5.1.1. Arithmetic operations. In this paper, we show the generating of stimuli for the arithmetic addition with carry operation. We can divide the process of creating production rules into several sections - *Input values*, *Logic*, and *Result*. Each section includes specific rules that are applied during the generation. The most complex section is *Logic* the production rules of which must ensure the correct procedure for calculating the result of this operation. The schematic representation of these sections is shown in Fig. 3 which shows also the parts of resultant stimulus.

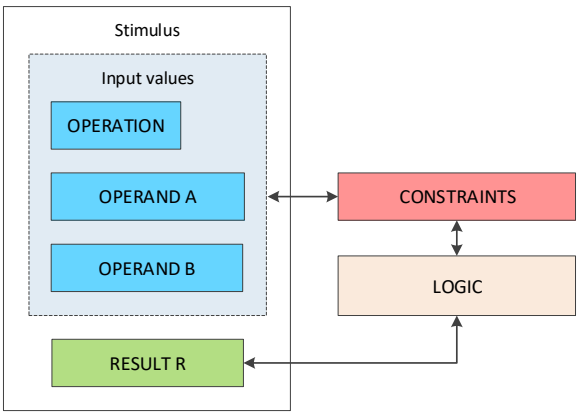


Fig. 3: The schematic representation of arithmetic operation in our framework.

As can be seen in the figure, stimulus is composed of four lines which are represented by integer binary numbers. The lines are generated sequentially as outlined, therefore, it is important to keep the context in which the rules were applied. The first line is the operation code followed by two operands (the numbers which are summed up) and the last line is a final result. The bit widths of inputs can be entered arbitrarily based on used ALU, e.g. for our ALU 1 bit can be long operation, 8 bits long operands, and 8 bits long result.

The constraints are also shown in the figure, because they are involved in the selection of production rules. Based on the random generation of input operands, certain constraints are set, and therefore, the logic is modified – the probabilities of production rules are deterministically set to produce an unambiguous result.

In the definition of production rules, each operand is divided into N non-terminals (N is equal to operand bit width). In our case, the operand A is divided into eight bit non-terminals A7-A0, where A7 is the most significant bit (MSB) and A0 is the least significant bit (LSB). The same applies for the operand B. The rules are as follows:

```
A -> A7 A6 A5 A4 A3 A2 A1 A0
B -> B7 B6 B5 B4 B3 B2 B1 B0
```

Each bit non-terminal A7-A0 can be zero or one, therefore, it can take one of the following two terminals (comma represents OR, terminals are in quotes):

```
A7 -> '0', '1'
A6 -> '0', '1'
…
A0 -> '0', '1'
```

Using these production rules, we have random value in the first operand. At the moment, we have not information about a carry bit propagation. The carry bit is determined during the generation of the operand B. For these purposes, it is necessary to keep the value of operand A. Therefore, each bit non-terminal B7-B0 can be replaced for non-terminal BiA0 (if Ai were zero), BiA1 (if Ai were one), BiA0C (if Ai were zero and a carry bit was set) or BiA1C (if Ai were one and a carry bit was set). These possibilities have to be reflected in production rules:

```
B7 -> B7A0, B7A1, B7A0C, B7A1C
B7A0, B7A1, B7A0C, B7A1C -> '0','1'
…
B0 -> B0A0, B0A1, B0A0C, B0A1C
B0A0, B0A1, B0A0C, B0A1C -> '0','1'
```

It remains to add production rules that will generate the final result:

```
R -> R7 R6 R5 R4 R3 R2 R1 R0
R8, R7, ..., R0 -> '0', '1'
```

Now it is known which values the input operands have and whether the carry bits have been propagated. These rules without any control would generate random non-terminals and the result would not reflect the operation addition with carry. Therefore, constraints have to be utilized. The framework performs the right derivations (substitution of the rightmost non-terminals) for the both operands and result, therefore, the substitution will start with the bit A0 to A7, then with B0 to B7, and then R0 to R7.

The B0 does not have a carry bit, therefore, we change the probability to zero for two rules with carry on the start of generation (S is the default starting non-terminal):

```
cons(->S, B0->B0A0C, 0);
cons(->S, B0->B0A1C, 0);
```

The context of the application of the rules for operand A have to be stored in operand B, therefore, we keep the context by limiting the selection of rules for operand B and its corresponding bit:

```
cons(A0->'0', B0->B0A1, 0);
cons(A0->'0', B0->B0A1C, 0);
cons(A0->'1', B0->B0A0, 0);
cons(A0->'1', B0->B0A0C, 0);
...
cons(A7->'0', B7->B7A1, 0);
cons(A7->'0', B7->B7A1C, 0);
cons(A7->'1', B7->B7A0, 0);
cons(A7->'1', B7->B7A0C, 0);
```

After this limitation, we have two rules for each bit B7-B1 which can be used after the generation of the operand A. The bit B0 have only one deterministic rule without the carry bit. After the generation of operand A and the bit B0, we are able to determine the carry bit (rule) for the following bit B1 and the result for bit R0. The same applies for the other bits B2-B6:

```
cons(B0A0->'0', R0->'0', 100);
cons(B0A0->'0', B1->B1A0C, 0);
cons(B0A0->'0', B1->B1A1C, 0);
cons(B0A0->'1', R0->'1', 100);
cons(B0A0->'1', B1->B1A0C, 0);
cons(B0A0->'1', B1->B1A1C, 0);
cons(B0A1->'0', R0->'1', 100);
cons(B0A1->'0', B1->B1A0C, 0);
cons(B0A1->'0', B1->B1A1C, 0);
cons(B0A1->'1', R0->'0', 100);
cons(B0A1->'1', B1->B1A0, 0);
cons(B0A1->'1', B1->B1A1, 0);
...
```

In this logic, constraints for rules `BiA0C` and `BiA1C` can be easily completed to obtain the correct result.

The selection of result bit after applying the rules is based on the following Tab. 1 which defines the classical addition with carry operation.

Tab. 1: Grammar truth table of addition with carry C.

Ai bit	Bi bit	Ri	Ci+1
Ai->'0'	BiA0->'0'	Ri->'0'	0
Ai->'0'	BiA0->'1'	Ri->'1'	0
Ai->'0'	BiA0C->'0'	Ri->'1'	0
Ai->'0'	BiA0C->'1'	Ri->'0'	1
Ai->'1'	BiA1->'0'	Ri->'1'	0
Ai->'1'	BiA1->'1'	Ri->'0'	1
Ai->'1'	BiA1C->'0'	Ri->'0'	1
Ai->'1'	BiA1C->'1'	Ri->'1'	1

The final real result can be seen as in the following example:

```
0                                   #OP
01101001                            #A
10001011                            #B
11110100                            #R
```

This process of creation is useful and usable for other arithmetic and bitwise operations. The main condition is to cover all possible cases (creation of corresponding production rules) which are then used or disabled by means of constraints during generation. The use of the constraints causes a fact that the defined grammar is more deterministic and the output is valid.

5.1.2. Bitwise operations. The process of creation grammar for the bitwise operations is very similar as in the previous subsection in the case of arithmetic operations. The basis is again to maintain the context through several production rules and their non-terminals. The difference is only in the generation of results, respectively the limitation of the rules for generating the partial bit of the result so that the output is correct for the given operation.

6. Experimental Results

We performed an experiment in functional verification in which we examined the highest coverage of the key functions of the presented ALU. Functional verification is the process of checking the correctness of a system based on comparing its inputs and outputs with reference model which implements the same specification. We had implemented verification environment in which we investigate the valid result of the ALU and the *code coverage*. The code coverage measures the system source code through typical metrics like statements, branches, expressions, conditions, and states. Through this information, we are able to determine, when the ALU is sufficiently verified. It is a percentage value suitable for comparison or different generators.

The result of our experiment can be seen in Fig. 4. From the experiment, it can be seen that there is a difference between our generator (USG) and the Build-in generator of test stimuli in verification environment. The both of the generators work on random stimuli construction but in our approach we are able to drive the generation process to direct the convergence to the better results. Verification environment checks also corner cases for input data (e.g. all ones or zeros in operands and result) and through probability values, we are able to increase the ability to generate this combinations. Therefore, the USG can hit this coverage points faster than only with clean random generation. The coverage was 94.91% for USG and 91.63% for Built-in generator for 100 stimuli. For 200 stimuli, the coverage was balanced for both generators on 94.91%.

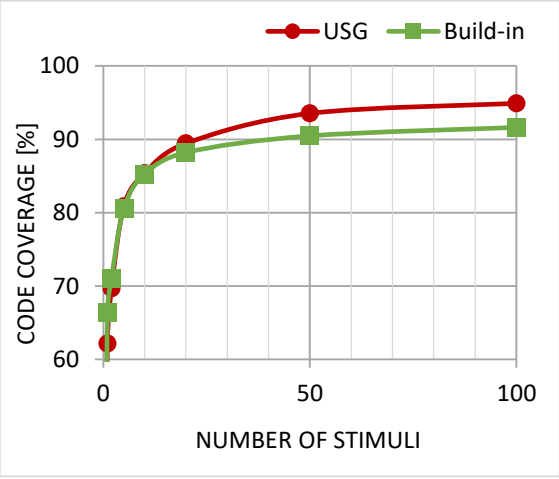


Fig. 4: The code coverage in functional verification.

7. Conclusions and Future Research

The aim of this paper was to show the possibility of generating as input as expected output. Automatic generation of random stimuli facilitates the work and time to test or verify a designed circuit. We showed on an arithmetic logic unit the generation of input and output together for which we defined our probabilistic constrained grammar. The output stimulus was composed of as randomly generated input operands as the expected result for this unit. The introduced mechanism has been shown on addition with carry operation, however, the defined principles are general and can be used for other arithmetic or bitwise operations, cyclic redundancy check generation, and so on. The experiment in functional verification showed that this principle is ductile to get better results than other ones.

This work is one of the partial goals for checking fault tolerance in Field Programmable Gate Array (FPGA). The main goal is to verify the correctness of affected system under a fault and to determine the importance of each of the configuration memory bits in FPGA. The future research will address this topic.

8. Acknowledgements

This research was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II); JU ECSEL Project SECREDAS (Product Security for Cross Domain Reliable Dependable Automated Systems), Grant agreement No. 783119; project IT4Innovations excellence in science - LQ1602 and BUT project FIT-S-17-3994.

9. References

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

[2] N. Kitchen and A. Kuehlmann. Stimulus generation for constrained random simulation. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 258-265, Nov 2007.

[3] R. Giegerich. *Introduction to Stochastic Context Free Grammars*. Humana Press, Totowa, NJ, 2014.

[4] O. Cekan,, J. Podivinsky, and Z. Kotasek. Program Generation Through a Probabilistic Constrained Grammar. In *2018 Euromicro Conference on Digital System Design (DSD)*, accepted to conference, 8 pages, Aug 2018.

[5] 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*, 52:145-159, 2017.

[6] D. A. Patterson. Reduced instruction set computers. *Commun*. ACM, 28(1):8-21, January 1985.

[7] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *Design and Test of Computers*, IEEE, 18(4):36-45, May 2001.

[8] M. Graphics. Verification academy - the most comprehensive resource for verification training, [Online] (2013). Available: www.verificationacademy.com.

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

[10] B. Wess. Automatic code generation for integrated digital signal processors. In *1991., IEEE International Sympoisum on Circuits and Systems*, pages 33-36 vol.1, Jun 1991.

[11] A. M. Amiri, A. Khouas, and M. Boukadoum. Pseudorandom stimuli generation for testing time-to-digital converters on an fpga. *IEEE Transactions on Instrumentation and Measurement*, 58(7):2209-2215, July 2009.

[12] G. Squillero. Microgp-an evolutionary assembly program generator. Genetic Programming and Evolvable Machines, 6(3):247-263, 2005.

[13] J. G. Bartkowiak and M. A. Nix. Arithmetic logic unit, Jan. 25 1994. US Patent 5,282,153.