Contents lists available at ScienceDirect

Microprocessors and Microsystems

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

Functional verification based platform for evaluating fault tolerance properties

Jakub Podivinsky*, Ondrej Cekan, Jakub Lojda, Marcela Zachariasova, Martin Krcma, Zdenek Kotasek

Brno University of Technology, Faculty of Information Technology, Centre of Excellence IT4Innovations, Bozetechova 2, 612 66 Brno, Czech Republic

A R T I C L E I N F O

Article history:
Received 9 January 2017
Revised 15 March 2017
Accepted 4 June 2017
Available online 12 June 2017

Keywords:
Functional verification
Robot controller
Electro-mechanical systems
Fault tolerance
Maze generation

A B S T R A C T

The fundamental topic of this article is the interconnection of simulation-based functional verification, which is standardly used for removing design errors from simulated hardware systems, with fault-tolerant mechanisms that serve for hardening electro-mechanical FPGA SRAM-based systems against faults. For this purpose, an evaluation platform that connects these two approaches was designed and tested for one particular casestudy: a robot that moves through a maze (its electronic part is the robot controller and the mechanical part is the robot itself). However, in order to make the evaluation platform generally applicable for various electro-mechanical systems, several subtopics and sub-problems need to solved. For example, the electronic controller can have several representations (hard-coded, processor based, neural-network based) and for each option, extendability of verification environment must be possible. Furthermore, in order to check complex behavior of verified systems, different verification scenarios must be prepared and this is the role of random generators or effective regression tests scenarios. Also, despite the transfer of the controller to the SRAM-based FPGA which was solved together with an injection of artificial faults, many more experiments must be done in order to create a sufficient fault-tolerant methodology that indicates how a general electronic controller can be hardened against faults by different fault-tolerant mechanisms in order to make it reliable enough in the real environment. All these additional topics are presented in this article together with some side experiments that led to their integration into the evaluation platform.

1. 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 is rising, 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 system operation.

The approach dealing with *design faults* is called *functional verification* [2] which currently has an irreplaceable position in the development cycle of digital systems. It runs in a simulation (RTL - *Register-Transfer Level* simulators are typically used, like QuestaSim from Mentor Graphics or VCS from Synopsys) and uses sophisticated testbenches which are prepared according to UVM (Universal Verification Methodology) [3,4] which ensures scalability and re-usability. 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 apply a huge number of input stimuli to the input ports of the verified circuit (DUT - *Device Under Test*) and on the input ports of the reference model. Afterwards, the behavior of DUT and the reference model is compared for these stimuli. The reference model is prepared by a verification engineer in SystemVerilog, C/C++ or other supported language and implements the reference behavior.

Coverage is an important metric in verification. It measures how well input stimuli cover the behavior of DUT and provides feedback that determines when the verification process can be ended. Depending on the coverage criterion considered, the following *coverage* metrics can serve as an example:

* Corresponding author.
E-mail addresses: ipodivinsky@fit.vutbr.cz (J. Podivinsky), icekan@fit.vutbr.cz (O. Cekan), ilojda@fit.vutbr.cz (J. Lojda), zachariasova@fit.vutbr.cz (M. Zachariasova), ikrcma@fit.vutbr.cz (M. Krcma), kotasek@fit.vutbr.cz (Z. Kotasek).

http://dx.doi.org/10.1016/j.micpro.2017.06.004
0141-9331/© 2017 Elsevier B.V. All rights reserved.

- *Code coverage* measures how well input stimuli cover the source code of DUT. Typical code coverage metrics are toggle, statement, branch, condition, expression or FSM coverage.
- *Functional coverage* measures how well input stimuli cover the functional specification of DUT. The user defines the coverage points for the functions to be covered in a verified circuit, e.g.: Did the verification test cover all possible commands or did the simulation trigger a buffer overflow?

Moreover, standard languages, methodologies and libraries were defined for functional verification. The most commonly known ones are the SystemVerilog IEEE language standard [5], Universal Verification Methodology and the open-source UVM library (with all the basic components of verification environments).

Of course, UVM-based functional verification does not guarantee 100% correctness of the system as formal verification does. The reason is that formal verification is based on an exhaustive exploration of the state space of DUT, hence it is potentially able to formally prove its correctness. However, the main disadvantage of this method is a state space explosion for real-world systems and the need to provide formal specifications of the behavior of the system which makes this method often hard to use. On the other hand, UVM-based functional verification is much easier to use and aims at covering properties determined by the specification, not the whole state space. Nevertheless, if these properties are selected accurately, all key aspects of the system are properly verified.

The approaches which deal with *hardware/physical* faults are techniques called *Fault avoidance* or *Fault tolerance* [6]. *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. Many fault-tolerant methodologies have been developed inclined, among others, to *Field Programmable Gate Arrays* (FPGAs) and new ones are under investigation [7], 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 [8] which are connected by programmable interconnections. The configuration is stored as a *bitstream* in SRAM memory. The problem is that FPGAs are quite sensitive to faults caused by charged particles [9]. This particle can induce an inversion of a bit in the 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 and some of them are performed on a theoretical level, for example, a simulation method for SEU emulation is presented in [10]. Another approach is in the use of fault injection directly into the design implemented in FPGA. Special evaluation boards are developed for these purposes, one of them is presented in [11] or [12].

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. Until now, our work was dedicated to verification of fault-tolerant qualities that allow us to check just the resilience of electronic components. However, 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.

This paper is organized as follows. The goals of our research are described in Section 2. Section 3 introduces three phases of the evaluation process based on our platform. We focus on introducing every phase theoretically and at the same time, we elaborate on making the platform general for various electro-mechanical systems. The first phase is mentioned in Section 4 together with verification environment architecture. Different possibilities for implementing the electronic controller (DUT) are mentioned in Section 5. This can be considered as the first step to generalization. The second step is preparing various verification scenarios for different DUTs and this process is summarized in Section 6. FPGA-based verification environment which is needed for the second phase is presented in Section 7. Principles which are used for checking reactions of the mechanical part in the third phase are introduced in Section 8. For the demonstration of our evaluation platform we created a case-study presented in Section 9 which is supplemented by experiments and their results in Section 10. Section 11 summarizes the results and proposes our plans for future research.

2. Goals of the research

Based on our previous analysis of actual research in the area of fault tolerance methodologies and their evaluation, we have identified the main goals that we would like to focus on in our research of fault-tolerant FPGA-based systems.

- The first point is to develop an *evaluation platform based on FPGA technology for testing fault tolerance techniques*. The basic concepts and the first version of the evaluation platform were presented in our previous work [13]. 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 important task is to propose the *process describing the use of the developed evaluation platform for fault tolerance properties improvement* in general electro-mechanical systems. It means that our evaluation platform will be supplemented with a description on how to configure the environment for the selected experimental system, especially how to evaluate fault tolerance properties and search for the possibilities of its improvement.
- As was mentioned above, we need to *take into account the mechanical part* which is usually driven by an electronic controller in real systems. Therefore, the verification environment should take into account also the operation of the mechanical part when evaluating the correctness of operations.

The following sections describe our progress in achieving these goals. Firstly, the basic concept of the evaluation process is described and is divided into three phases. Each of these phases needs a specific verification environment with a specific configuration, so the evaluation platform is described on a theoretical level for every phase separately. The evaluation platform is demonstrated on one case-study: a robot searching a path through a maze and its electronic controller.

3. Basic concept of the evaluation process

The proposed process of the fault impact evaluation, which is shown in Fig. 1, is divided into three phases. In the first phase, we use the simulation-based functional verification where the VHDL description of the electronic controller is used as the DUT. In this phase, the correctness of the electronic controller design is evaluated. The main output of the first phase is a test on whether the electronic controller works correctly according to the specification. It is important because we have to ensure that the electronic controller does not contain any functional errors in the implementation. It is also important to point out that in this phase we acquire

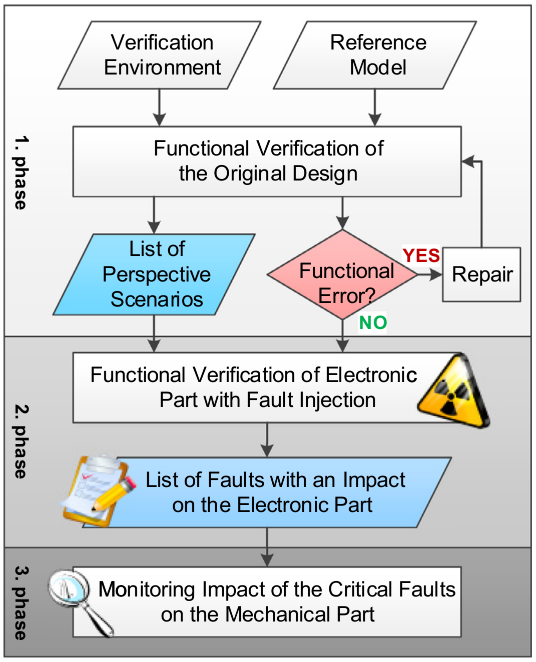

Verification Environment
Reference Model
Functional Verification of the Original Design
1. phase
List of Perspective Scenarios
Functional Error?
YES
Repair
NO
Functional Verification of Electronic Part with Fault Injection
2. phase
List of Faults with an Impact on the Electronic Part
3. phase
Monitoring Impact of the Critical Faults on the Mechanical Part

Fig. 1. The flow of phases in the FT evaluation systems verification.

a set of verification scenarios that will also be used in the subsequent phase.

The second phase consists of the verification of the electronic controller implemented into FPGA with the scenarios obtained during the previous phase, but in addition, artificial faults are injected into FPGAs using implemented fault injector.

The analysis of the faults which corrupted the mechanical part is the goal of the third phase. 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).

The development of the verification environment and a reference model for the electronic control unit (the electronic controller) are the first steps towards this whole three-phase process. Both of these activities are described in detail in this paper. The second step is to implement the DUT into the FPGA and achieve interconnection with the simulation environment of the mechanical part. The architecture of the verification environment with the electronic controller implemented in the FPGA is also presented in this paper.

4. The first phase - verification environment architecture

The verification environment architecture, its basic components and used techniques are described in this section. First, the UVM based verification environment for one verification scenario (one model of real environment, one task for electro-mechanical system) is presented, which forms the core of an extended verification environment for multiple verification scenarios evaluation.

The verification environment for the electronic controller is designed according to UVM, so that it corresponds with current trends and requirements. The basic architecture of the verification environment with main components is shown in Fig. 2. It should be noted that the verification environment is connected with the

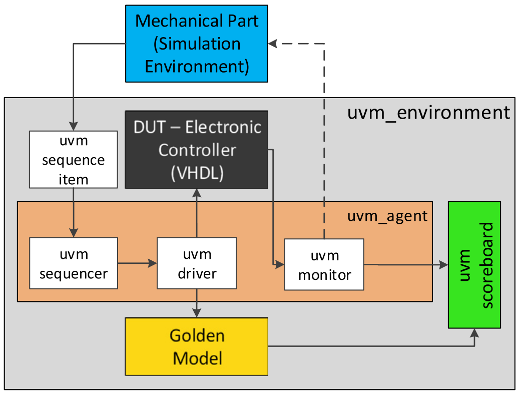


Fig. 2. General verification environment for a single verification scenario.

mechanical part, especially the simulation of the mechanical part. The mechanical part in real environment is controlled by the outputs of the electronic controller (DUT) while the outputs of the mechanical part (information from sensors) are inputs for the electronic controller. The information whether the 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 electronic controller* under a verification which can be implemented into FPGA in the next phase. Several approaches how to implement DUT exist, they will be described in Section 5.
- *The golden (reference) model* implemented in C/C++ according to the same specification as the electronic controller performs the same operations as DUT.
- *The sequence* is the component which receives data from sensors placed in the mechanical part. Received data are transformed to the inputs of the verification environment.
- *The driver* sends input values (data from sensors) to reference model and the DUT (electronic controller).
- *The monitor* reads the outputs from the DUT (instructions for mechanical part) and forwards them to the scoreboard and to the mechanical part which operates according to these values.
- *The scoreboard* compares the outputs of the monitor and reference model for equality and checks mismatches on the outputs. The detected mismatch shows that there are differences between the DUT and the reference model outputs.

The presented verification environment is not able to evaluate multiple verification scenarios automatically so it must have been extended by components such as verification scenarios random generator or a simulator of the mechanical part. All the final components, their inputs, outputs and connections are shown in Fig. 3 and their description is as follows:

- *The verification scenarios generator* allows us to generate a sufficient number of verification scenarios with respect to specified parameters in order to achieve the required coverage. In our work, we use a verification scenario generator based on our universal generating principle described in Section 6.
- *The mechanical part simulation* replaces the real mechanical part in case we do not have a real one.
- *The UVM verification environment* is the core of the extended evaluation.
- *Store the verification scenario* allows us to use it in the second phase which utilizes a fault injector (Fig. 1). A certain part of the stored verification scenario is also a report about the coverage which was obtained by this scenario.

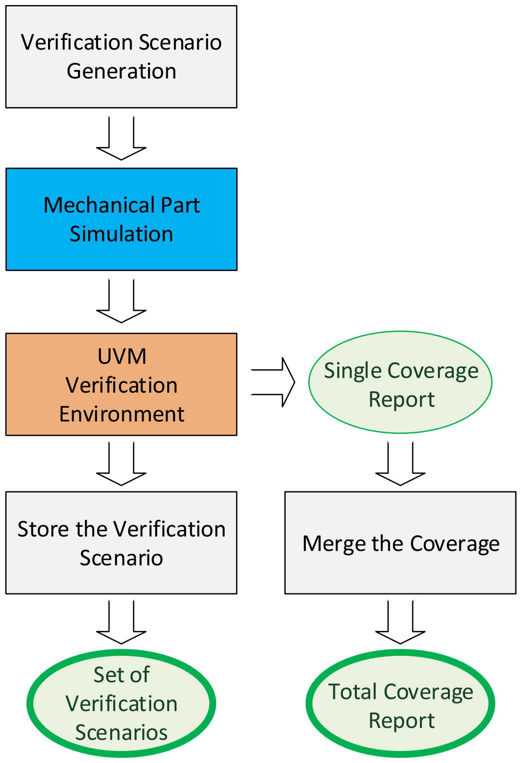

Verification Scenario Generation
Mechanical Part Simulation
UVM Verification Environment
Single Coverage Report
Store the Verification Scenario
Merge the Coverage
Set of Verification Scenarios
Total Coverage Report

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

- *Merge the coverage* achieved by the single verification scenario is important in order to obtain a final coverage report gained by stored sets of verification scenarios.

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

5. Electronic controller - implementation alternatives

As mentioned above, an electronic controller can be implemented in several ways, each having different advantages and disadvantages. Several key features exist which must be taken into account. From one point of view it can be flexibility, scalability and extensibility. Another point of view may be fault tolerance, durability and maintainability. From the economic point of view it is the cost, power consumption and time to market (the difficulty of development).

In this paper we mention three possibilities of controller implementation - a processor, a hard-coded controller and a controller based on neural networks.

5.1. The use of processor as electronic controller of mechanical part

The usage of a processor as the controller is the most universal and flexible way of controlling the system. A lot of different classes and types of processors are in the market and it is possible to find a suitable device for the application. The main advantages are the flexibility of the solution due to the possibility of changing the software, short time to market and the low cost (if the processor is selected properly).

The usage of a processor also offers different ways to build a fault-tolerant controller. The processor can be secured using hardware redundancy - more processors can be used to build a robust system, or the processor can be secured on the level of its inner components. Another way is to secure the application at the software level using time and space redundancies [14].

5.2. Hard coded electronic controller

In this case, the controller is composed of components described using a hardware description language (HDL). This solution is less flexible than the previous one, but it can be more efficient. The hard coded controller can be designed directly to the application in order to utilize resources effectively which can lead to high performance and/or low power consumption. High performance can be reached due to the possibility of utilizing high parallelism and application-designed specific computing units. The effective computing algorithms can then be implemented. The low power consumption is related to the effective usage of resources. Unneeded components can also be omitted (unlike in case of processor).

Several ways on how to ensure fault tolerance exist in this case. Replicating the whole system is the well known way, however in the case of hardcoding, it is easier to construct the redundancy on the level of inner components - the computing units, registers, multiplexers and other components the system is composed of. This makes it possible to secure only selected components, therefore, making the fault-tolerant design more effective by avoiding the redundancy where it is not needed. The component specific techniques can be deployed as well. It is also possible to use more specific techniques like securing using partial TMR [15]. It is also possible to use different coding schemes for securing the data - for example, parity codes, BCH codes and others error detection/correction codes. If the FPGA is used, the reconfiguration can offer suitable tools for fault-tolerant techniques implementation [16].

This solution can generally be very effective in relation to performance, power consumption and fault tolerance. However, the time to market is longer and costs can be higher than in case of using the processors. Nevertheless, this solution can be suitable for tasks with specific recommendations.

5.3. The use of neural networks as fault-tolerant electronic controller

Artificial neural networks are one of the traditional disciplines in the field of artificial intelligence and softcomputing. Even though the neural networks were almost forgotten for some period of time, at present their popularity is continually growing. The main advantage of neural networks is their capability of learning and memorizing the data which make them suitable for different tasks such as classification, function approximation, timeseries prediction, etc. They are widely used, especially in the deep form which disposes of very interesting properties for the tasks such as image recognition. The neural networks are also used in control tasks [17], the control of mobile robots included [18,19]. Therefore, we are going to experiment with a neural-network-based controller as well.

The neural networks dispose of interesting fault-tolerant properties which are used with different techniques. It is possible to modify a learning algorithm to train the network, not only to perform the task, but to be fault-tolerant as well [20]. It is also possible to retrain the network after a fault occurs [21]. Other approaches use adding an extra redundancy into the network [22].

In our research we are dealing with the specific FPGA resource saving implementation of neural networks called FPNN [23]. We especially deal with its neural network approximation capabilities [24] and also with designing fault-tolerant techniques. In this manner, we designed the fault-tolerant type of FPNNs [25].

6. Verification scenarios generation

A very important part in the verification process is preparing and applying input stimuli. Just by using a significant number of diverse inputs, it is possible to cover most of the behavior of DUT and thus to be sure that DUT behaves as specified. When we consider the standard approach, stimuli are represented by transactions in the UVM-based verification environments. Transaction stands for a setting of input ports of DUT in one clock cycle. So, for example, when DUT has three input ports A (8 bits), B (16 bits), C (1 bit), the transaction can be represented by a triplet of values {8'h87,16'h11FF,1'b0} which is applied on these input ports on the rising edge of synchronization clock signal (or other specified synchronization event).

Two approaches for preparing input stimuli exist:

- Directed stimuli - a verification engineer prepares transactions manually. This approach is recommended at the beginning of the verification process for checking basic scenarios.
- Pseudo-randomly generated stimuli - transactions are generated by a random generator according to the given constraints (restrictions for values of inputs). That is the reason why this approach is often called constraint-based generation. An example of constraining an input is the following. Let us have an input port A (8 bits), then a constraint can restrict the generated values for this port in the range of 0–100 (from the possible range 0–255).

In the following text, we will introduce the pseudo-random generation of stimuli using our proprietary generator. This generator is unique in the way that it can be used for different scenarios and works with two formal models that significantly improves its performance. Afterwards, we will show how an evolutionary algorithm can be used for preparing optimal regression suites for different systems. The motivation for using regression suites is simple. If there is a need for running verification of DUT repeatedly - just to check that everything still works or after minor changes to DUT (like small optimizations), it is worth using an optimized small suite of tests (input stimuli) with a high level of DUT coverage rather than start verification from scratch every time (it is very time consuming).

The same applies to the evaluation platform. Depending on the electronic controller that is verified, we can either utilize direct stimuli, pseudo-randomly generated stimuli or an effective regression test suite (especially when we want to evaluate fault injection in the second and third phases of our evaluation process introduced in Section 3).

6.1. Pseudo-random generation of verification scenarios

Pseudo-random generation is also an integral part of our research. We are creating the solution for the stimuli generation which has to be versatile for our evaluation platform. We need a different stimuli for a different system that we verify. The original way of the generation that we presented in our previous paper [26] is generalized by using probabilistic context-free grammars [27] that we found as a suitable means for the stimuli generation. We gained universal description of verification scenarios (stimuli) while maintaining our originally designed architecture. We benefit from grammar systems which allow us to generate a defined language. This language will form stimuli for a given system. We are also able to control the generation process through the defined probabilities in the grammar. In our architecture, we use constraints which allow us to modify stimuli during their generation and verification.

In our previous research [26], we proposed our solution of the generator which was based on our own proprietary description of the stimuli. Although the designed architecture was general, the description of the stimuli was not versatile for any system. A specific dependencies in stimuli creation had to be implemented when a new system should be supported. About 13 types of the constraints had to be implemented for valid generating of assembly programs for a RISC processor. The set of the constraints also increased with the inclusion of support for other systems. For these reasons, we were looking for another suitable solution that will be built on any mathematical apparatus. As the best solution, we have found the use of the grammatical system [28] from the theoretical computer science.

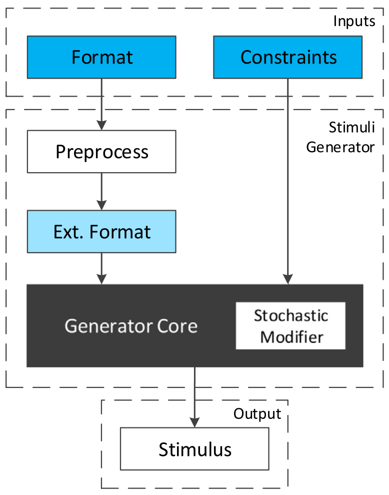


Fig. 4. The detailed architecture for a probabilistic context-free grammar based stimuli generation.

In our actual research, we use the probabilistic context-free grammars. The probabilistic context-free grammar is the quintuplet:

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

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

The probabilistic context-free grammar looks like a common context-free grammar, but it has the special set of probabilities which represent how likely a rewrite rule of the grammar is applied. It allow us to define the format of the stimuli through the formal description provided by grammars. We benefit from the probability definition for the rules, because it allows us to control the application of the rules in the grammar and gives us the possibility to influence the stimuli creation. The probabilities are defined statically for the grammar definition which is not optimal for stimuli creation. Therefore, we extended this description with a special logic that is described through the constraints which allow us to dynamically change defined probabilities.

The constraints represent restrictions and limitations for the application of the rewrite rules of the grammar and their use will change defined probabilities for specific rules during generation process such that the result is a valid stimulus. After application of any rule, the eligible constraints are performed and the new probabilities are set anywhere in the grammar.

The architecture of the generation is shown in Fig. 4. The probabilistic context-free grammar is defined in the input structure called *Format*, while special rules for restricting the grammar are defined in the *Constraints* structure. For an easier description of the input structure, we use certain elements of the library Jinja2 [29] which is a templating system for the Python programming language [30]. The templating system allows us to define cycles, branches and other special macros in the structure description. The preprocessing (*Preprocess*) expands these special macros and a complete description of stimuli (*Ext. Format*) is obtained. The extended format suffices to be generated only when the original format is changed, otherwise, the generator works directly with this extended format and is not generated any more.

The Ext. Format and the Constraints 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. Probabilities are adjusted using a special block *Stochastic Modifier*. Through this, the next derivation valid for the given stimulus is directed and prescribed constraints will be respected for generating a valid one.

6.2. Regression test suites optimization using evolutionary algorithms

Our optimization technique was firstly introduced in paper [31] and later on extended in paper [32]. It works off-line and takes a suite of input stimuli that were evaluated in the process of UVM-based functional verification and optimizes them automatically using the genetic algorithm (one of the evolutionary algorithms). The aim of optimization and the main contributions of this technique are:

1. *Eliminating the redundancy* in the original suite of stimuli so the optimized suite is smaller and therefore, it will be running faster in simulation.
2. *Preserving the same level of coverage* (the term coverage was explained in Section 1) as was achieved by the original (unoptimized) suite of stimuli. It guarantees that the behavior of DUT will be checked properly.
3. *Reusing already created verification environment for running regressions* after minor changes in DUT are made so it is not necessary to utilize a separate approach for regression testing.

Redundancy in the original suite of stimuli is caused by their randomness. In the first phase of verification when DUT is firstly created, redundancy in stimuli is often a beneficial factor [33], because key properties of the system have a chance to be checked more times (for example, we want to check multiplication, but it is good to check it repeatedly with different data) and it is almost always wanted. But after this phase, for example during regression testing, the redundancy is not needed anymore (it is enough to check every key property of the system just once), so it is good to have regression stimuli that are effectively reduced from the original suite and thus are running faster (in order to spend less time by running regressions). That is the reason why we decided to apply our optimization after the first phase of verification with the aim to reduce redundancy.

The survey of the proposed optimization technique follows; the process of optimization is divided into several steps:

1. Run the UVM-based functional verification for a selected DUV and collect stimuli until the threshold in coverage is reached.
2. Optimize stimuli by the proposed technique.
3. Use the optimized suite everytime regression testing is needed. It is even possible to use existing verification environment for running regressions.

The optimization technique incorporates the genetic algorithm as the main optimization tool. As described in [34], the genetic algorithm employs a population of candidate solutions that are evolved through several generations. The quality of candidate solutions is determined by the *fitness function*. According to the fitness, the best solutions are selected and serve as parents for the next generation. Offsprings are created by mutation and crossover genetic operators. If the algorithm works well, the average fitness of the population is rising because profitable parts of the search space are explored. At the same time, genetic operators ensure diversity, so the algorithm is resilient to the problem of local optimum.

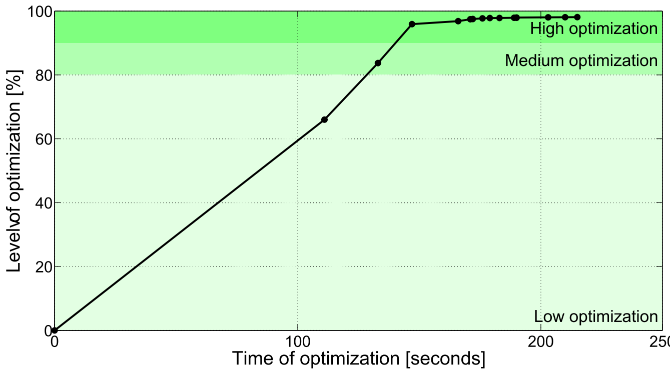


Fig. 5. The dependency between the optimization runtime and the level of optimization.

The main result is that the presented optimization technique was able to reduce the number of stimuli to the *0.522% of the original size* and the resulting coverage statistics remained at the same level as was achieved by the original suite. What is more important, the simulation runtime of the optimized regression suite was much shorter and was reduced by *98.1%*. See more details in paper [32].

Fig. 5 demonstrates the dependency between the achieved level of optimization of the regression suite (the y axis in the graph) and the time of optimization (the x axis in the graph).

It can be seen that the longer the optimization runs, the shorter is the simulation runtime for regression testing.

7. The second phase - evaluation platform architecture

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

1. software part of verification environment for the electronic controller running on a computer,
2. software simulation environment for mechanical part simulation running on a computer,
3. electronic controller implemented into FPGA, and
4. external fault injector [35] running on a computer which allows us to simulate real faults in the FPGA.

The overall experimental platform interconnection is shown in Fig. 6. 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 below.

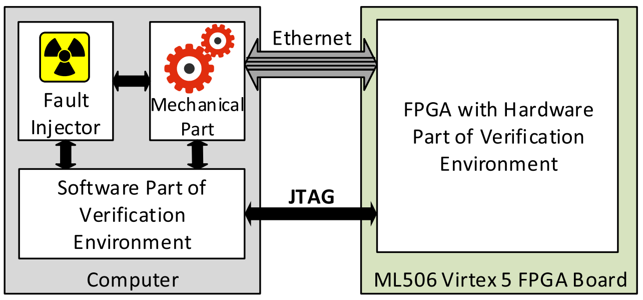

Fault Injector
Mechanical Part
Software Part of Verification Environment
Computer
Ethernet
JTAG
FPGA with Hardware Part of Verification Environment
ML506 Virtex 5 FPGA Board

Fig. 6. The structure of the experimental platform.

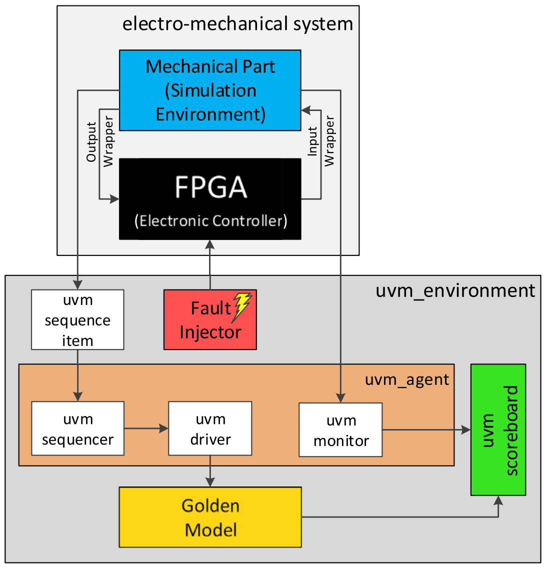


Fig. 7. The general architecture of the FPGA-based verification environment.

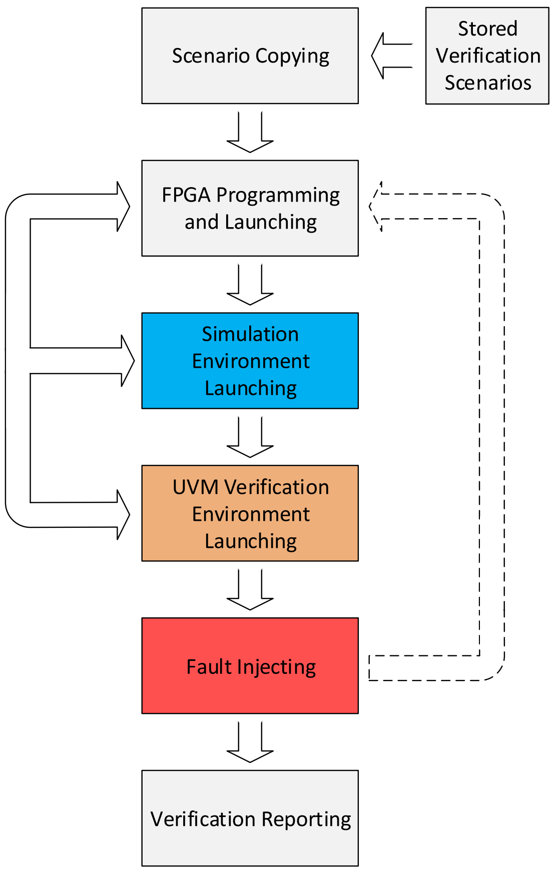

Scenario Copying
Stored Verification Scenarios
FPGA Programming and Launching
Simulation Environment Launching
UVM Verification Environment Launching
Fault Injecting
Verification Reporting

Fig. 8. FPGA-based verification environment for multiple evaluation with fault injection.

7.1. Architecture of FPGA-based verification environment

For these purposes, the FPGA-based verification environment, which is displayed in Fig. 7, 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 mechanical part which is controlled by the electronic controller implemented into the FPGA. The communication between the software and the hardware parts is accomplished using a proprietary interface (more details about the communication are provided in the subsequent subsections). This part operates autonomously, and the electronic controller receives information from the sensors which is 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 electronic controller implemented in the FPGA to the mechanical part 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 electronic controller and a mechanical part 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 Fig. 7. Information from the sensors is received in the Sequence component where they are transformed to transactions and transferred to the Golden Model which produces reference output data. Instructions for mechanical part are received in the Monitor component and sent to the Scoreboard component. The Scoreboard compares received data with reference data obtained from the Golden Model.

Both parts are synchronized by signals sent from the Sequence and Monitor components to the mechanical part simulation environment. These signals indicate that the verification environment is ready to observe operations of mechanical part.

The presented FPGA-based verification environment evaluates only one verification scenario, but automated evaluation of multiple verification scenarios with a fault injection is needed which is shown in Fig. 8. The second phase eliminates the need for verification scenarios generation because scenarios pregenerated and verified in the first phase are used. Conversely, there are new steps as a consequence of implementing electronic controller into the FPGA and the creation of an autonomous connection between the FPGA and the mechanical part. The first necessary step is programming the FPGA through the JTAG interface which must be done before each verification run. This step ensures that the correct functionality of the electronic controller is verified and is without faults.

The next step is launching the mechanical part into a simulation and verification environment which enables signals to the simulation environment when it is ready to start monitoring. Then, the mechanical part starts its operation 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 has expired and then results of the verification are recorded into the verification report.

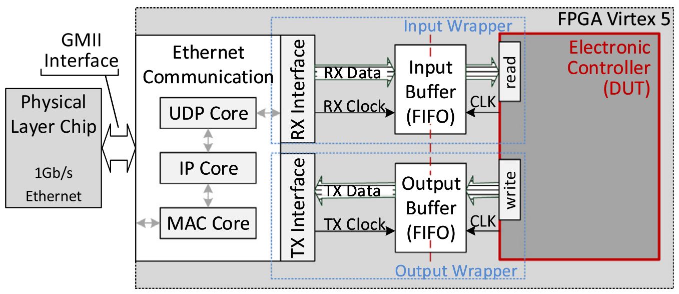


Fig. 9. The architecture of communication between SW and HW part.

During fault injection, it is worth utilizing effective regression test suites for experiments. There are two reasons for this.

The first reason is that regressions contain stimuli that achieve a high level of total coverage (the coverage of DUT behavior is very high) and, therefore, after fault injection, we can be sure that such stimuli/tests discover all potential problems combined with injected faults regarding functionality. To be precise, it is guaranteed for DUT that if no artificial faults are injected, it always behaves correctly for regression tests. After a fault is injected and an error or errors occur, it only means that the fault caused a critical problem inside the system. The result of this phase is a list of faults and their locations, which caused discrepancy on the outputs of DUT for a specific regression stimulus/test. Furthermore, it is possible to run an advanced analysis and harden some critical parts of DUT by fault-tolerant techniques, and to check the results for the selected regression tests again, until we are satisfied with the result.

The second reason is that regressions are usually running much faster in RTL simulation as they represent significantly filtered pseudo-randomly generated stimuli. And when we consider running verification after every single injected fault, the time-saving is significant.

7.2. Communication between software and hardware parts

Communication between the software and hardware parts of verification environment can be accomplished in various ways. One way is the use of some proprietary interface, or another way is to use one of the standardized interfaces which are used in verification based on emulation in FPGAs [36]. One of them is Standard Co-Emulation Modelling Interface (SCE-MI) [37] proposed by the Accellera organization. Thanks to the standard SCE-MI interface, users are able to reuse the existing hardware cores in FPGA in order to develop their system prototype.

In our case we chose to use Ethernet interface supplemented with our proprietary protocol based on UDP. The communication between the electronic controller implemented in the FPGA (hardware part) and the mechanical part in a simulation environment (software part) is accomplished through Input and Output Wrapper. We chose the ML506 development board [38] equipped 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). The chip implementing the Ethernet physical layer is connected to the FPGA and to the 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, instead we use an existing implementation presented in [39].

Fig. 9 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 the phase offset. We use a FIFO memory as an input and output buffer with different writing and reading clock signals. This not only solves the problem with clock domain crossing, but also the problem with data storing before their processing. Data received from the 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 the DUT which allow us to exchange a communication layer with another one which uses FIFO buffers.

7.3. 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 developed a fault injector [35] 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 bitstreams are 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 the RapidSmith [40] tool. This tool identifies the bits of bitstream which are related to a specified area in the FPGA. Functional units placement in the FPGA is done by the PlanAhead [41] tool, so we know where each of the functional units is 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).

8. The third phase - mechanical part reactions

The task for the third phase is to check reactions of the mechanical part and there are several methods on how to do it. We can look either at the mechanical part or on its simulation and check whether it works. These methods require an observer, which can be a person or a digital camera supplemented with some kind of an algorithmic image processing.

Another way is to use a type of information which represents the state of mechanical part. In modern electro-mechanical systems, there are lots of sensors placed on a mechanical part which informs us about its state. These sensors are usually used as an input for the electronic controller, and we can use the information from these sensors for monitoring the behaviour of the mechanical part. For example, if we are monitoring the movement of an autonomously driven car, we can observe its behaviour by monitoring the GPS location and a speed sensor.

Our evaluation platform is based on functional verification which only observes the electronic outputs of verified circuits implemented into FPGA. These electronic outputs are compared with the outputs of the reference model which operates as a part of the

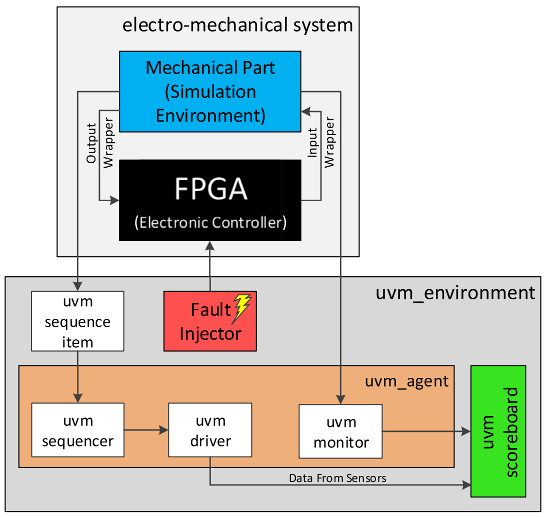


Fig. 10. Checking of the mechanical part behaviour by functional verification.

verification environment according to the same specification as the verified circuit. Input for the reference model is the information from sensors produced by the mechanical part which is the same as for the verified circuit. Values from sensors received by the verification environment can be used not only as inputs for the reference model, but also as inputs for monitoring behaviour of the mechanical part. The modified verification environment is shown in Fig. 10 where the reference model is missing, information from sensors is routed directly to the scoreboard which can monitor the operation of the mechanical part. It means that scoreboard implements functions for checking reactions of mechanical part based on information from sensors.

9. Casestudy: robot searching a path through a maze and its electronic controller

General principles used in our platform were presented in previous sections and our experimental electro-mechanical system will serve as a demonstration example in the following text. As an experimental system we chose a robot which searches for a path through the maze. The mechanical part is a robot in the maze and the electronic part is its electronic controller. Unfortunately, we do not have a real robot device, so we used simulation environment for a robot and its environment. We use Player/Stage [42] simulation environment which is freely available and offers lots of possibilities for robot configuration.

Our robot is a simple cubical robot which goes through the maze. The robot is equipped with a few sensors, three sensors which inform us about distances from three control points placed at fixed positions in the robot environment. They are used for determining its location (inspired by Global Positioning System). Four sensors are located on the sides of the cubical robot and inform us about the distances from barriers in the robot surrounding. The operation of the robot is driven by two inputs - speed of robot in x-axis and y-axis directions (*x_speed, y_speed*).

The robot controller, whose structure is shown in Fig. 11 consists of various blocks, their function is described in [43]. The controller is connected to the PC on which the 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 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 the information about the distance to the surrounding barriers. The map updating provided by the Map Unit (MU) is based on the 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. The 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 a bus is accomplished through the bus wrapper (FU_WB) and controlled by the finite state machine (FU_FSM).

Our electro mechanical system was introduced, but verification environments for three phases of verification process must also be proposed. These verification environments are implemented according to principles presented above.

9.1. Simulation based verification environment (the first phase)

The general verification environment for the first phase shown in Fig. 2 is a standard UVM-based functional verification environment which is usually created during electronic systems development. In our example, the electronic controller is a robot controller and the mechanical part is a robot going through a maze. The reference model is implemented with respect to the same specification as the robot controller, the inputs are the information from sensors and outputs are speed values in x-axis and y-axis directions. These speeds are compared with the speed values received from the robot controller. Naturally, the compared speed values must be the same. The verification environment and reference model are presented in more details in [44].

The verification environment is able to process multiple verification scenarios (see 3), while one verification scenario is, in the case of robot, represented by a maze and start and goal positions of the mission. Therefore, stimuli generation in this casestudy means mazes generation.

9.2. Maze generation

Maze generation is a well known and explored area for which a considerable number of algorithms generate simple or sophisticated mazes [45]. The vast majority of algorithms operate in a two-dimensional space, keeping their 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 generation, 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 such 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 Fig. 12. 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. In order to maintain the continuity from any corner of the maze to another, it is necessary to perform a 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 western side of the maze. The final and most critical task is to determine cells A, B, C, D which allows us to have the maximal continuous maze (c).

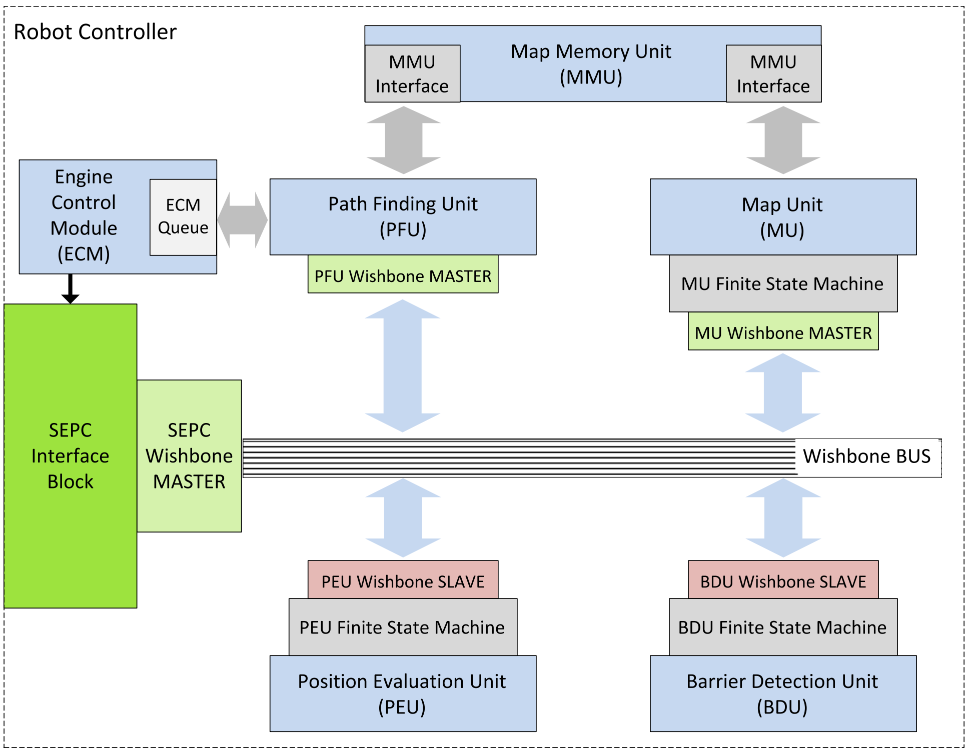


Fig. 11. The block diagram of the robot controller.

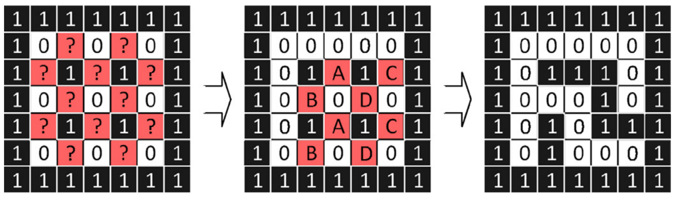


Fig. 12. The demonstration of a conversion of the basic matrix of the maze for needs of the generator.

The original description of the algorithm [45] 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 western 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 cell A, respectively C, was randomly selected for the corner in Fig. 12.b, then cell B, respectively D, will be a wall and vice versa.

9.3. FPGA-based verification environment (the second phase)

The second phase of our verification process is based on the FPGA-based verification environment. The environment is shown in Fig. 13. It can be seen that the electronic controller is represented by the robot controller implemented into FPGA and the mechanical part is represented by a robot in a maze simulated in the Player/Stage environment for robot simulation. The verification environment just observes the communication between the robot in the maze and its robot controller.

Multiple verification scenarios evaluation is done with respect to the process shown in Fig. 8. Stored verification scenarios are mazes with start and goal positions saved during the previous phase. The important step in this process is fault injection which allow us to inject faults according to various strategies.

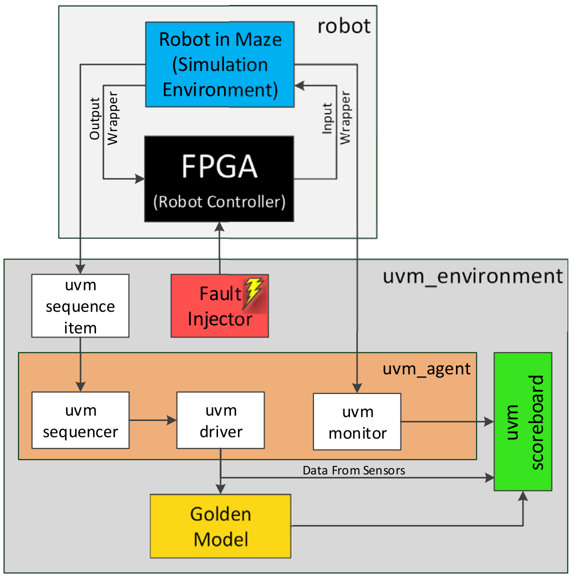


Fig. 13. The architecture of the FPGA-based verification environment for the robot controller.

9.4. Mechanical part reactions (the third phase)

Checking behaviour of the mechanical part is done by monitoring the information provided by sensors on a robot. The distances from three control points are used for monitoring robot trajectory through the maze, especially checking if the robot finds a goal position. The information about barriers are used for the detection of collisions with a wall. The information about barri-

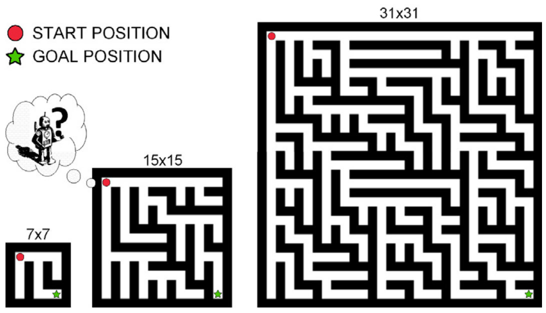


Fig. 14. Three types of mazes.

Table 1
Average number of robot steps.

Maze size	7 × 7	15 × 15	31 x 31
Average number of steps	16	93	433

ers are represented by four values with distances from the wall in four-neighbourhoods of the robot in the maze. These values can be compared with predefined minimal values and verification can detect if the robot is closer to the wall or if the robot crashes into the wall.

Fig. 10 shows general functional verification environment for the third phase. The values from sensors are routed directly to the scoreboard and this verification environment is dedicated just to checking behavior of the mechanical part. In our example, we created one combined verification environment which serves both for the second and the third phase. This verification environment is shown in Fig. 13 where values from sensors and values from a reference model are inputs to the scoreboard which checks electronical and mechanical parts concurrently. It means that scoreboard implements functions both for checking outputs of robot controller and also for checking reactions of mechanical robot. Behavior of mechanical robot is checked by monitoring distances of robot to the wall.

10. Casestudy: experiments and results

Performed experiments correspond to the activities of all phases of the fault tolerance evaluation process.

10.1. The first phase - simulation based verification

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. Fig. 14 shows three types of mazes which were used in our experiments. The presented mazes differ in their dimensions and we chose 7x7, 15x15 and 31x31 cells. Examples of start and goal positions are also shown in Fig. 14. With the growing size of the maze the number of steps that the robot must go through increases. The average number of the robot steps in various types of mazes is shown in Table 1. The main goal of the experiments, 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.

For the experiment, we chose the number of performed verification scenarios equal to 10, 100, 200 and 500, for which we monitored an achieved code coverage. The numbers of performed verification scenarios were the same for all types of mazes and in total 1500 verification scenarios were performed with a variety of mazes. Various bugs were identified and debugged during the verification process. It can be stated that the robot controller operates according to its specification for the performed 1500 verification scenarios.

Experimental results are presented in Table 2. It can be recognized that the maximal achieved total code coverage is 91.85%. The inability of achieving an ideal 100% is caused by the default 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. a 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 7 x 7 to 15 x 15 cells led to a slight increase of code coverage, which is possibly due to the effect of the maze. When increasing the size of maze to 31x31 cells, the coverage was not changed. Such studies show that the 7 x 7 cells maze is too small for the next phase of fault impact evaluation process. This trend is shown in the bar chart in Fig. 15 which shows the code coverage for different sizes of mazes for 100 verification scenarios (the part of Table 2).

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.

10.2. The second phase - controller reactions

The second phase in the proposed evaluation process is targeted towards evaluating the correct function of a robot controller implemented into the FPGA. For this purpose fault injection is used. No fault tolerance methodology implemented in the robot controller for these experiments was used and the goals of the experiment were: 1) detailed reliability analysis of the robot controller and its functional units, and 2) a demonstration that the evaluation platform can be used for a fault tolerance evaluation.

As was mentioned above, faults can be injected in a way which reflects various strategies. Similar experiments were done in our previous work [13], 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 mazes of larger dimensions, the mazes of 15 x 15 for this phase. The robot controller consists of 15 functional units which leads to 750 verification runs

Table 2
The experimental results.

# of verification scenarios	10			100			200			500		
Size of mazes	7 x 7	15 x 15	31 x 31	7 x 7	15 x 15	31 x 31	7 x 7	15 x 15	31 x 31	7 x 7	15 x 15	31 x 31
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%

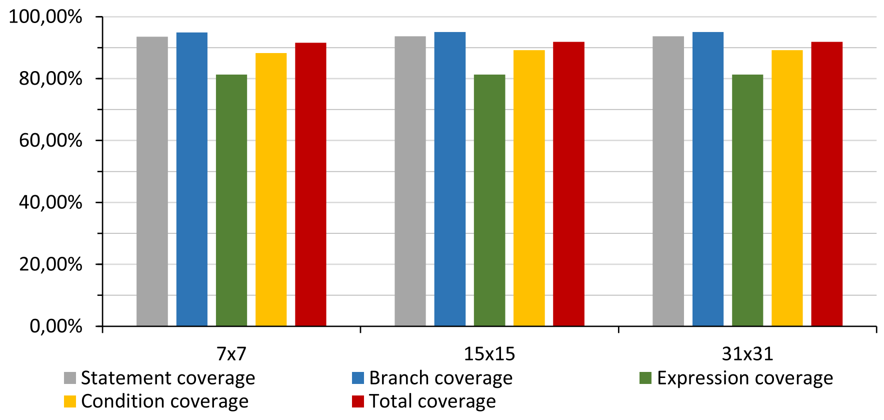


Fig. 15. Code coverage for each type of mazes for 100 verification scenarios.

Table 3
Experimental results in functional verification.

Unit	Number of fails	Fails in %	Unit	Number of fails	Fails in %
bdu	40	80.00	mu_wb	32	64.00
bdu_fsm	20	40.00	peu	39	78.00
bdu_wb	35	70.00	peu_fsm	40	80.00
ecu	38	76.00	pfu	34	68.00
intercon	30	60.00	pfu_wb	28	56.00
mmu	31	62.00	sif_fsm	50	100.00
mu	25	50.00	sif_wb	34	68.00
mu_fsm	1	2.00			

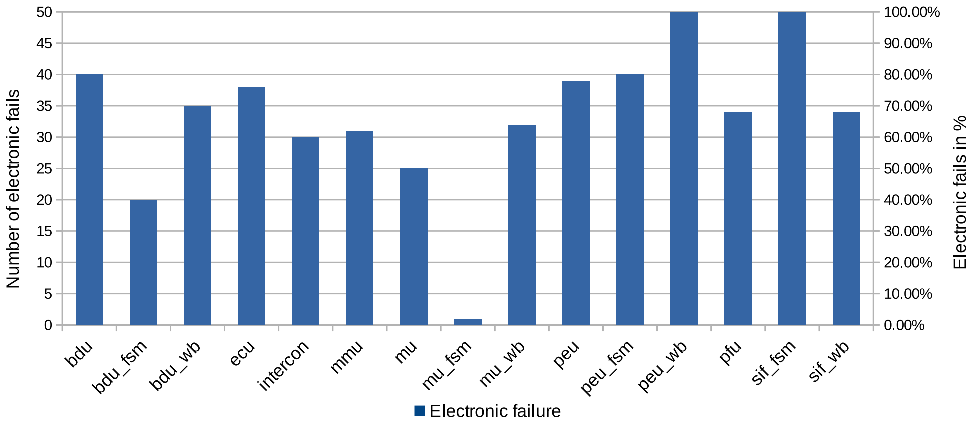


Fig. 16. Experimental results in functional verification.

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 3 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 Fig. 16 as well. The bar chart expresses a percentage number of faults with their impact on the robot controller. As can be seen, some anomalies in the results of the experiments 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. The *peu_fsm* and *sif_fsm* functional units represent a completely different situation, the number of faults with an impact that 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 4 shows additional verification runs that were performed in order to analyse these anomalies in detail. As can be seen, the additional results are closer to the overall average.

Table 4
Extended experimental results.

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

10.3. The third phase - mechanical part reactions

The evaluation of mechanical robot behaviour was the main task for the third phase. In this phase fault injection is also used. Faults are injected according to the same strategy as in the second phase because the second and the third phases share the

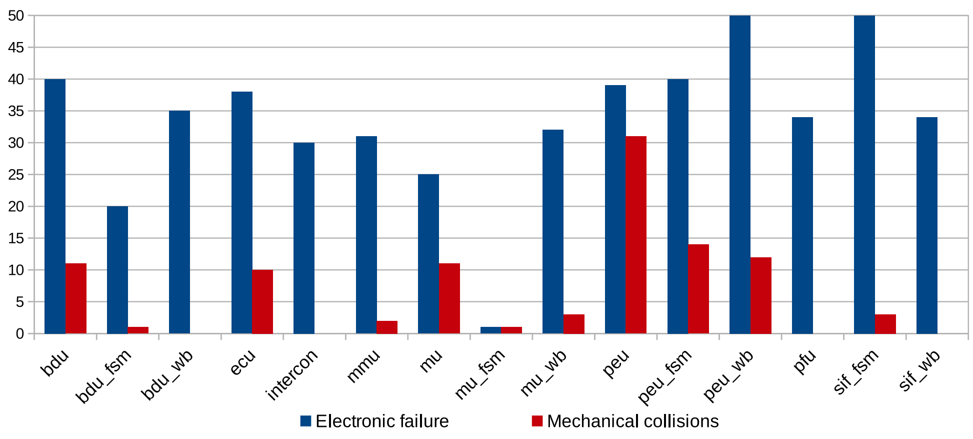


Fig. 17. Experimental results in functional verification.

Table 5
Number of robot collisions.

Unit	Number of el. fails	Number of collisions	Collisions in %
bdu	40	11	27.50
bdu_fsm	20	1	5.00
bdu_wb	35	0	0.00
ecu	38	10	26.32
intercon	30	0	0.00
mmu	31	2	20.00
mu	25	11	4.00
mu_fsm	1	1	22.00
mu_wb	32	3	2.00
peu	39	31	6.00
peu_fsm	40	14	62.00
pfu	34	12	28.00
pfu_wb	28	0	0.00
sif_fsm	50	3	6.00
sif_wb	34	0	0.00

same verification environment. The robot controller was also used without fault tolerance methodologies application and the goal of the experiments corresponding to the third phase was 1) to show the most frequent incorrect behavior of mechanical part; and 2) a demonstration that the evaluation platform based on functional verification is able to monitor the behaviour of the mechanical part.

In these experiments, we found that the most common consequences of injected faults are robot stopping at one place and robot collision with a wall. We can say that all verification runs with electronic failure leads to one of these consequences. If the electronic failure leads to the robot stopping at one place, it usually does not cause any damage. But on the other hand, the collision of the robot with the wall can lead to economical losses. Table 5 summarizes the number of electronic failures for each component and this information is supplemented by the number of collisions with the wall. Also, the number of percentage of electronic failure which lead to a collision is shown.

These results are also presented in the graphical version in the bar chart shown in Fig. 17. It is evident that the percentage number of collisions is different for each functional unit. It shows that some of functional units are more important to robot navigation than others. For example, the percentage for *peu_fsm* functional unit is the highest and the main task of this unit is routing information about its position to the bus. The path through the maze is searched by *pfu* and the movement of the robot is controlled by *ecu*, so the percentage number of collisions corresponding to these two functional units is also quite high.

We have made a fault injection analysis of the robot controller. We 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 other units. On the other hand, the units *peu_fsm* and *sif_fsm* are the most prone units for faults. A failure of electronic part usually leads to the robot stopping at a place and to its collision with a wall. As was mentioned above, some of the damaged functional units lead to collisions in more cases than others. So, these functional units are more problematic from a safety point of view. This analysis is especially important for future applications of fault-tolerant methodologies. A system designer obtains the information on which blocks need more attention from a reliability point of view.

The second finding is that we are able to use functional verification in conjunction with the fault injector to determine the impact of faults on the electro-mechanical system. If fault tolerance methodologies will be applied to the electro-mechanical system (in our case, the robot controller) our platform would be used to monitor impact of faults on system hardened against faults and therefore to automate the evaluation of these fault tolerance methodologies.

11. Conclusions and future research

In this work, we presented our evaluation platform for testing fault-tolerant methodologies and evaluation impact of faults on correct operation of electro-mechanical systems. Our evaluation platform is based on functional verification where the verified circuit is running on FPGA which allows us to inject faults directly to the FPGA. Our evaluation process is divided into three phases and each of these phases needs a specific verification environment. Firstly, a basic verification environment was introduced for the first phase of the evaluation process which is able to evaluate a single verification scenario and the creation of an extension that allows automated evaluation of multiple verification scenarios which were presented as well. This automated evaluation uses the verification scenarios produced by our generator or those which are part of the optimized regression test suite. The verification environment for the second phase where DUT is implemented to the FPGA was also mentioned. In the proposed methodology, the verification environment acts as an observer that checks data transferred between the electronic and mechanical parts. During the last phase, the impact of faults to the mechanical part is monitored by checking values received from sensors placed on a robot in a maze. The verification environment for the third phase was also introduced.

For a demonstration of our evaluation platform we proposed one demonstration example which uses a robot and its electronic controller as an experimental electro-mechanical application. Performed experiments correspond to all phases of a fault impact evaluation process. The output of the first phase was the debugged electronic controller and the list of verification scenarios for the next phase. During the concurrent second and third phase, the reliability analysis was done by means of fault injection into the FPGA. The result was the ratio of faults that caused an incorrect output of the electronic controller and the number of faults that caused the robot collision.

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 mentioned in this paper. We can also use more conventional fault-tolerant methodologies, such as TMR, on-line 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 621439 (ALMARVI) and BUT project FIT-S-17-3994.

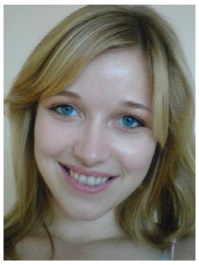
References

[1] A. Benso, P. Prinetto, Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Frontiers in Electronic Testing, vol. 23, Springer Science & Business Media, 2003.
[2] A. Meyer, Principles of Functional Verification, Elsevier Science, 2003 http://books.google.cz/books?id=qaIiX3hYWL4C.
[3] A. standard, Universal Verification Methodology, 2016 http://www.accellera.org/downloads/standards/uvm.
[4] V.R. Cooper, Getting Started with UVM: A Beginner's Guide, Austin, TX:Verilab, 2013.
[5] IEEE Standard for Systemverilog - Unified Hardware Design, Specification, and Verification Language, 2005, doi:10.1109/IEEESTD.2005.97972.
[6] I. Koren, C.M. Krishna, Fault-Tolerant Systems, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
[7] F. Siegle, T. Vladimirova, J. Ilstad, O. Emam, Mitigation of radiation effects in SRAM-based FPGAs for space applications, ACM Comput. Surv. 47 (2) (2015) 37:1–37:34, doi:10.1145/2671181.
[8] XILINX, FPGA, 2014 http://www.xilinx.com/fpga/index.htm.
[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, vol. 50, 2003, pp. 2088–2094.
[10] C. Bernardeschi, L. Cassano, A. Domenici, 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.
[11] M. Alderighi, S. D'Angelo, M. Mancini, 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.
[12] 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: Defect and Fault-Tolerance in VLSI Systems, 2007. DFT'07. 22nd IEEE International Symposium on, IEEE, 2007, pp. 105–113.
[13] J. Podivinsky, O. Cekan, M. Simkova, 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.
[14] O. Goloubeva, M. Rebaudengo, M.S. Reorda, M. Violante, Software-Implemented Hardware Fault Tolerance, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
[15] H.R. Mahdiani, S.M. Fakhraie, C. Lucas, Relaxed fault-tolerant hardware implementation of neural networks in the presence of multiple transient errors, IEEE Trans. Neural Netw. Learn. Syst. 23 (8) (2012) 1215–1228, doi:10.1109/TNNLS.2012.2199517.
[16] L. Miculka, Z. Kotasek, Generic partial dynamic reconfiguration controller for transient and permanent fault mitigation in fault tolerant systems implemented into FPGA, in: Design and Diagnostics of Electronic Circuits Systems, 17th International Symposium on, 2014, pp. 171–174, doi:10.1109/DDECS.2014.6868784.
[17] M. Chen, R. Mei, Actuator fault tolerant control for a class of nonlinear systems using neural networks, in: Control Automation (ICCA), 11th IEEE International Conference on, 2014, pp. 101–106, doi:10.1109/ICCA.2014.6870903.
[18] Z. Li, Fault diagnosis and fault tolerant control of mobile robot based on neural networks, in: Machine Learning and Cybernetics, 2009 International Conference on, vol. 2, 2009a, pp. 1077–1081, doi:10.1109/ICMLC.2009.5212376.
[19] Z. Li, Application of fault tolerant controller based on RBF neural networks for mobile robot, in: Intelligent Ubiquitous Computing and Education, 2009 International Symposium on, 2009b, pp. 531–534, doi:10.1109/IUCE.2009.140.
[20] B.S. Arad, A. El-Amawy, On fault tolerant training of feedforward neural networks, Neural Netw. 10 (3) (1997) 539–553, doi:10.1016/S0893-6080(96)00089-5.
[21] C. Sequin, R. Clay, Fault tolerance in artificial neural networks, in: Neural Networks, 1990., 1990 IJCNN International Joint Conference on, 1990, pp. 703–708 vol.1, doi:10.1109/IJCNN.1990.137651.
[22] Z.-H. Zhou, S.-F. Chen, Z.-Q. Chen, Improving tolerance of neural networks against multi-node open fault, in: Neural Networks, 2001. Proceedings. IJCNN '01. International Joint Conference on, . 3, 2001, pp. 1687–1692 vol.3, doi:10.1109/IJCNN.2001.938415.
[23] B. Girau, FPNA: Concepts and Properties, in: A.R. Omondi, J.C. Rajapakse (Eds.), FPGA Implementations of Neural Networks, Springer US, 2006, pp. 63–101, doi:10.1007/0-387-28487-7-3.
[24] M. Krcma, J. Kastil, Z. Kotasek, Mapping trained neural networks to FPNNs, in: Design and Diagnostics of Electronic Circuits Systems (DDECS), 2015 IEEE 18th International Symposium on, 2015a, pp. 157–160, doi:10.1109/DDECS.2015.50.
[25] M. Krcma, Z. Kotasek, J. Kastil, Fault tolerant field programmable neural networks, in: Nordic Circuits and Systems Conference (NORCAS): NORCHIP International Symposium on System-on-Chip (SoC), 2015, 2015b, pp. 1–4, doi:10.1109/NORCHIP.2015.7364381.
[26] J. Podivinsky, O. Cekan, M. Simková, Z. Kotásek, The evaluation platform for testing fault-tolerance methodologies in electro-mechanical applications, Microprocess. Microsyst. 39 (8) (2015) 1215–1230, doi:10.1016/j.micpro.2015.05.011.
[27] R. Giegerich, Introduction to Stochastic Context Free Grammars, Humana Press, Totowa, NJ, 2014, doi:10.1007/978-1-62703-709-9_5.
[28] A. Meduna, Formal Languages and Computation: Models and Their Applications, first, Auerbach Publications, Boston, MA, USA, 2014.
[29] A. Ronacher, Jinja2 (the python template engine), 2014 http://jinja.pocoo.org/.
[30] M. Lutz, Learning Python, second, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.
[31] M. Belesova, M.Simkova, Z.Kotasek, T.Hruska, Application of evolutionary algorithms for regression suites optimization., in: IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, IEEE Computer Society, 2015, pp. 91–949.
[32] M. Belesova, M.Zachariasova, Z.Kotasek, Regression test suites optimization for application-specific instruction-set processors and their use for dependability analysis., in: Proceedings of the 19th Euromicro Conference on Digital Systems Design, IEEE Computer Society, 2016, pp. 380–387.
[33] B. Wille, J. Goss, W. Roesner, Comprehensive Functional Verification, Elsevier, 2005.
[34] T. Bäck, J. Kok, Handbook of Natural Computing, Springer-Verlag, Berlin Heidelberg, 2012.
[35] 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.
[36] C.-Y. Huang, Y.-F. Yin, C.-J. Hsu, T.B. Huang, T.-M. Chang, SoC hw/sw verification and validation, in: 16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011), IEEE, 2011, pp. 297–300.
[37] Accellera, Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual, 2011 http://accellera.org/images/downloads/standards/sce-mi/SCEMIv21110112final.pdf.
[38] Xilinx Inc., Ml506 Evaluation Platform User Guide, UG347 (v3. 1.2), 2011.
[39] P. Skibik, Implementation of Ethernet Communication Interface into FPGA Chip, Technical Report, 2011 https://www.vutbr.cz/wwwbase/zavpracesouborverejne.php?fileid=40494.
[40] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, B. Hutchings, Rapid prototyping tools for FPGA Designs: RapidSmith, in: Field-Programmable Technology (FPT), 2010 International Conference on, 2010, pp. 353–356, doi:10.1109/FPT.2010.5681429.
[41] N. Dorairaj, E. Shiflet, M. Goosman, PlanAhead software as a platform for partial reconfiguration, vol. 55, 2005, pp. 68–71.
[42] 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.
[43] 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.
[44] S. Krajcir, Functional Verification of Robotic System Using UVM, Technical Report, 2015 http://www/study/DP/DP.php?id=15154.
[45] J. Buck, Maze generation: algorithm recap, 2011 http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap.

Jakub Podivinsky was born in 1989. In 2013 he graduated (MSc) at the Department of Computers Systems of the Faculty of Information Technology, Brno University of Technology. In 2013 he started his PhD 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 (MSc) at the Department of Computers Systems of the Faculty of Information Technology, Brno University of Technology. In 2013 he started his PhD studies at the Department of Computers Systems. His scientific research is focused on functional verification and stimuli generation.

Jakub Lojda was born in 1991. In 2015 he graduated (MSc) at the Department of Computers Systems of the Faculty of Information Technology, Brno University of Technology. In 2015 he started his PhD studies at the Department of Computers Systems. His scientific research is focused on fault-tolerant systems design automation.



Marcela Zachariasova was born in 1987. In 2011 she graduated (MSc) at the Department of Computers Systems of the Faculty of Information Technology, Brno University of Technology. In 2011 she started her PhD studies at the same university and successfully finished in 2015. The topics of her PhD thesis and the main areas of interest are optimization of UVM-based functional verification, automated verification of processors and fault-tolerant system design.

Martin Krcma was born in 1988. In 2014 he graduated (MSc) at the Department of Computers Systems of the Faculty of Information Technology, Brno University of Technology. In 2014 he started his PhD studies at the Department of Computers Systems. His scientific research is focused on fault tolerant implementations of artificial neural networks in FPGAs.

Zdenek Kotasek was born in 1947. He received his MSc. and PhD. degrees (in 1969 and 1991) from Brno University of Technology (BUT), both in computer science. Between 1969 and 2001, he worked at the 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, he was in the position of the DCSY head since 2005 till 2015. 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 senior member (since 2015).