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 Block*s (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.

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).

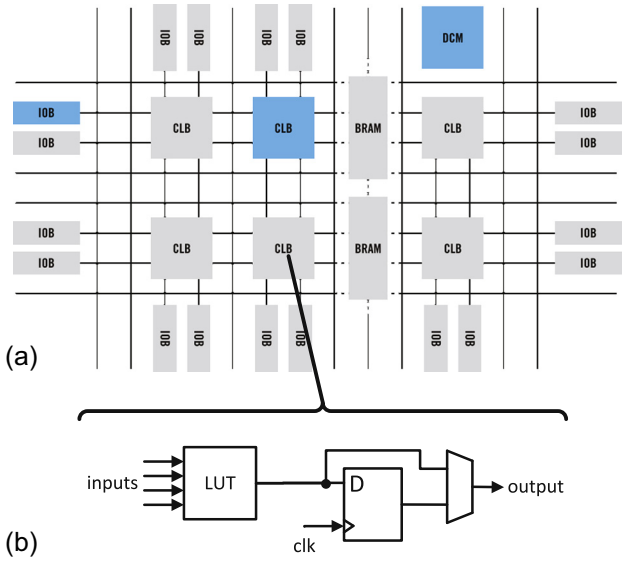


Fig. 1. Structure of (a) *Field Programmable Gate Array* (FPGA) and (b) *Configurable Logic Blocks* (CLB).

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

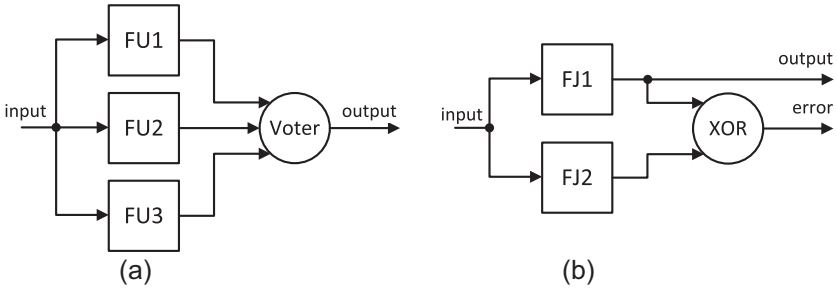


Fig. 2. Architectures with hardware redundancy: (a) TMR and (b) the duplex architecture.

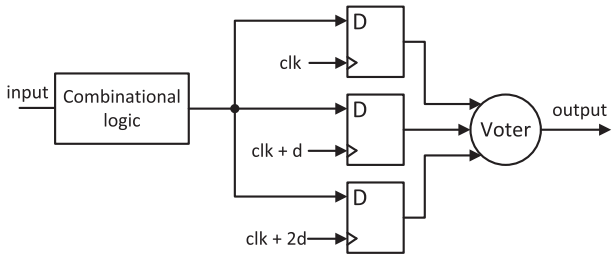


Fig. 3. Time redundancy basic scheme for combinational logic.

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

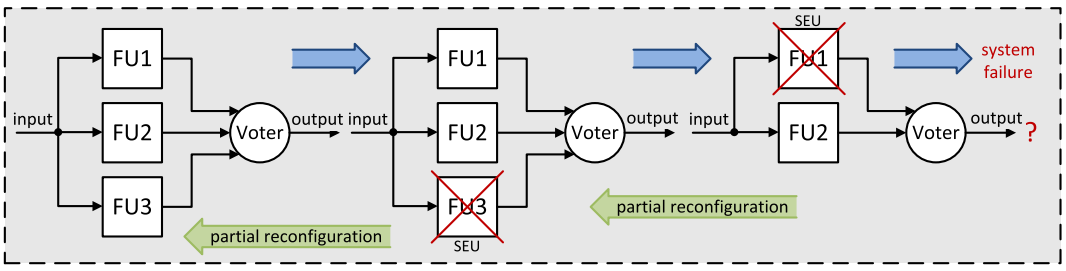


Fig. 4. Recovery of the faulty module in TMR by PDR.

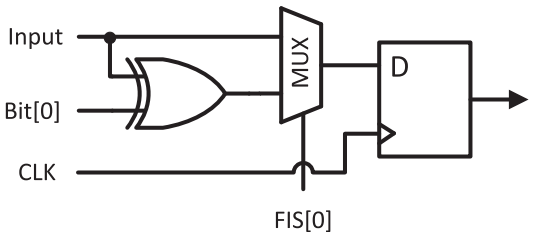


Fig. 5. The synthetizable SEU model [18].

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 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.

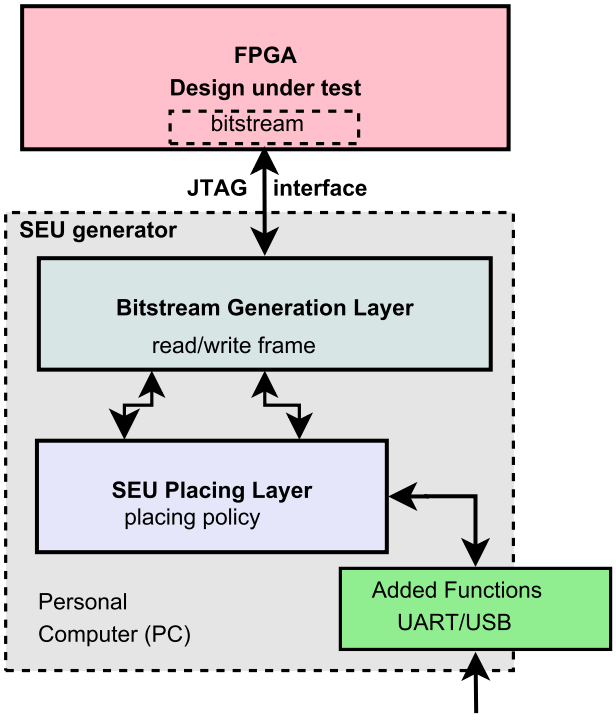


Fig. 6. An external SEU injector structure [22].

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.

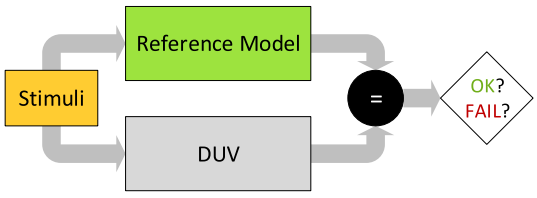


Fig. 7. The main principle of functional verification.

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

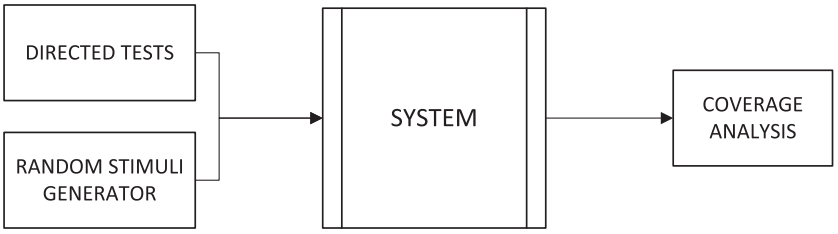


Fig. 8. The method with random stimuli generator.

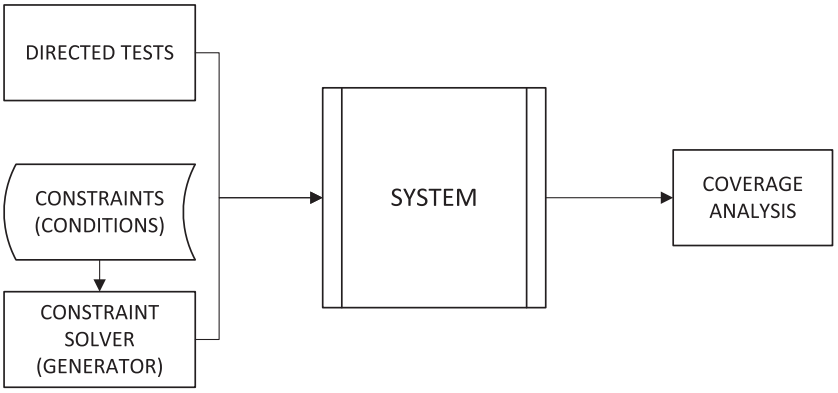


Fig. 9. The method with constrained-random stimuli generator.

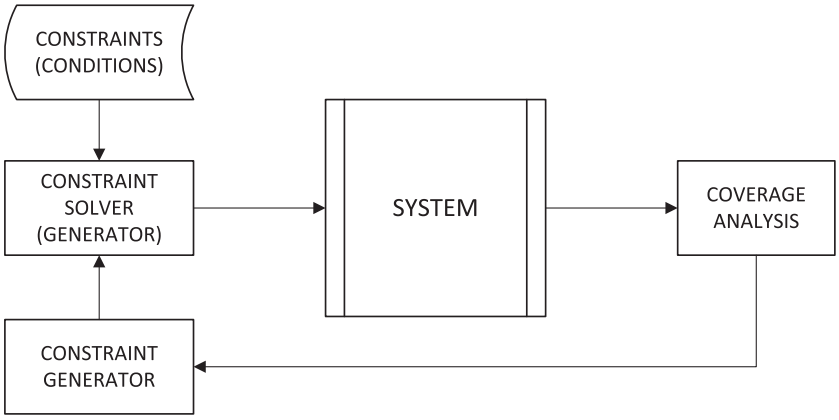


Fig. 10. The method coverage directed stimuli generation.

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.

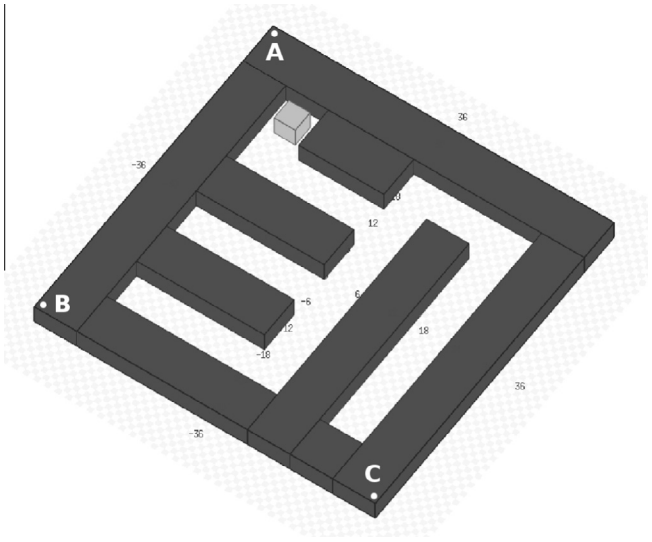


Fig. 11. The robot in a maze in player/stage simulation environment.

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.

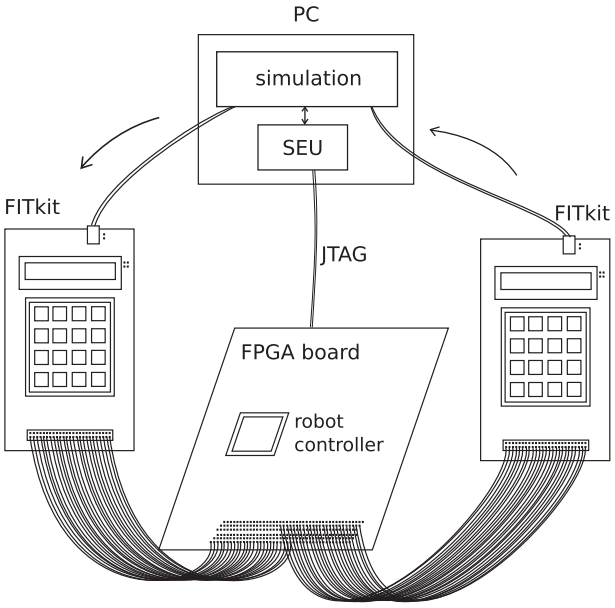


Fig. 12. The platform for testing fault-tolerance methodologies.

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.

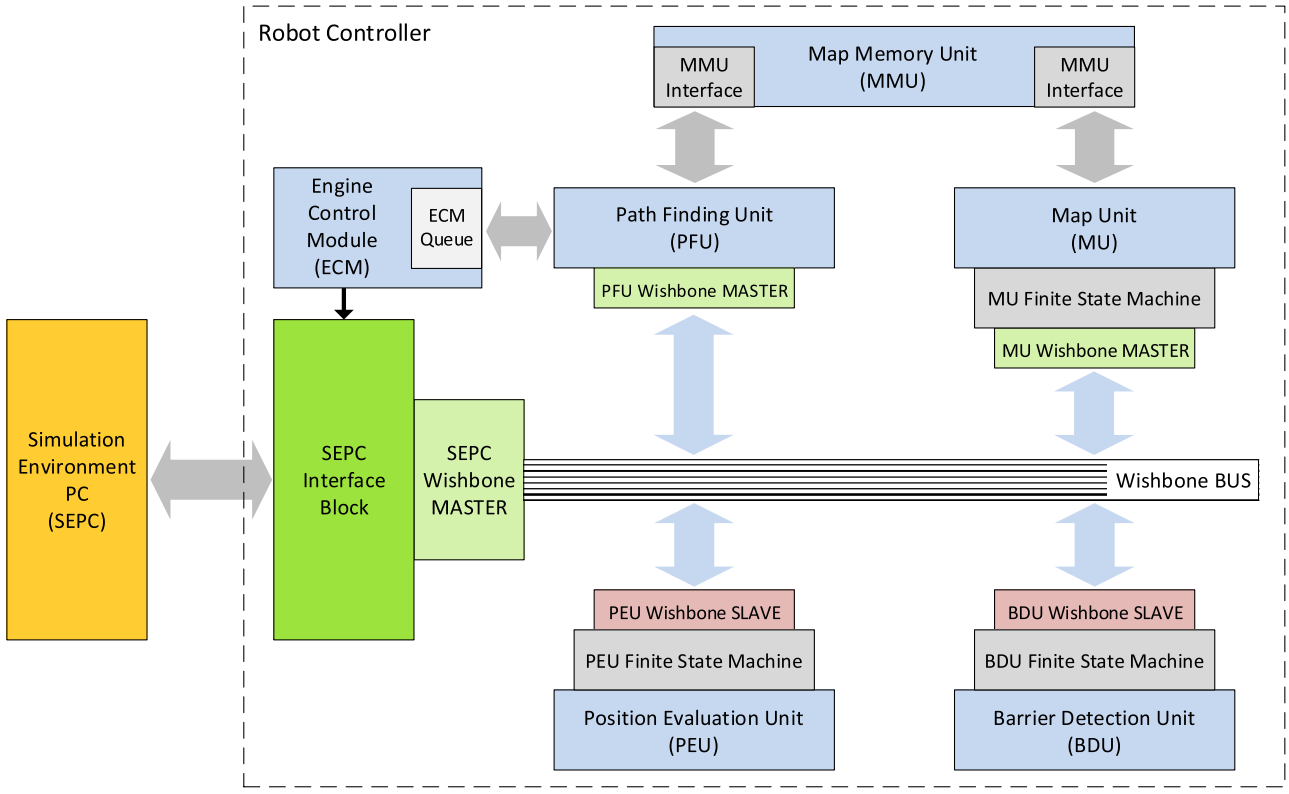


Fig. 13. The block diagram of the robot controller.

- *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

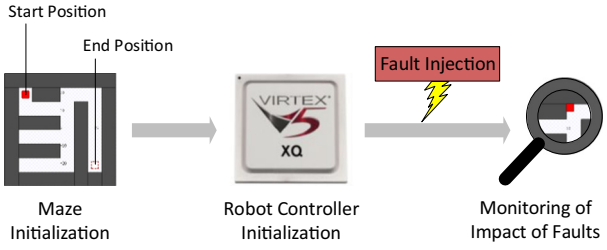


Fig. 14. The flow of one experiment.

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

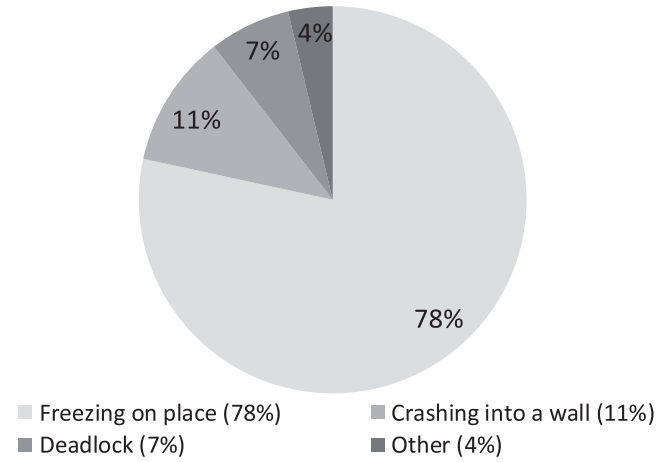


Fig. 16. The chart of typical consequences of injected faults on the mission of the robot.

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

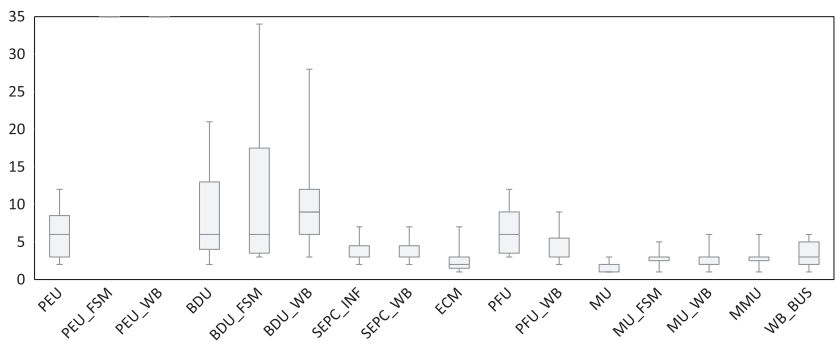


Fig. 15. The quartile graph of the results of experiments.

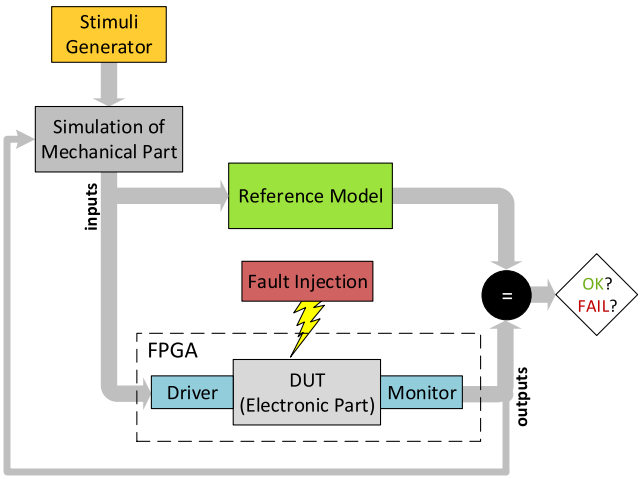


Fig. 17. The functional verification involvement in our platform with the fault injection.

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.

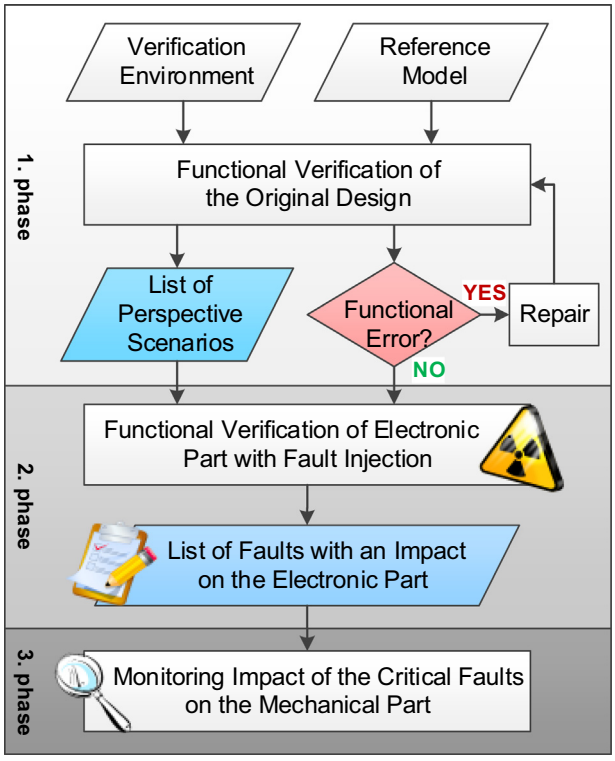


Fig. 18. The flow of phases in the FT systems verification.

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

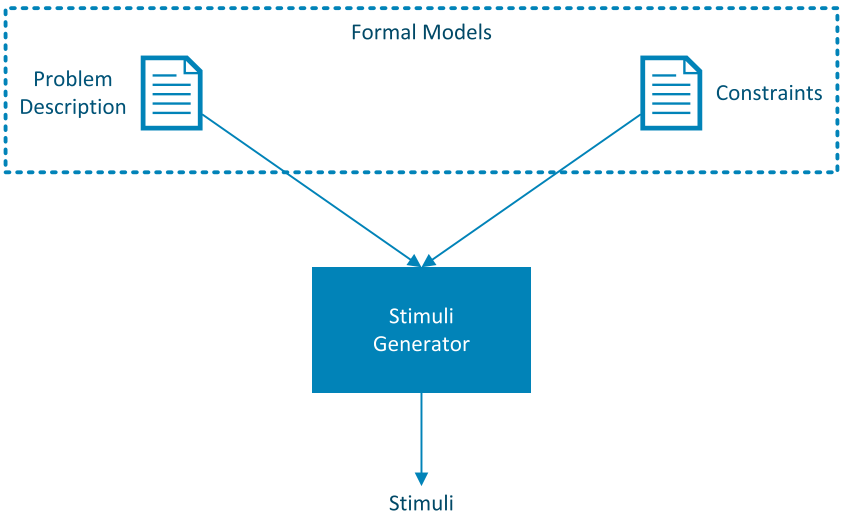


Fig. 19. The architecture of the stimuli generation.

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 {
  synnamel, 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 {
  repll, repl2, ...{subsl|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

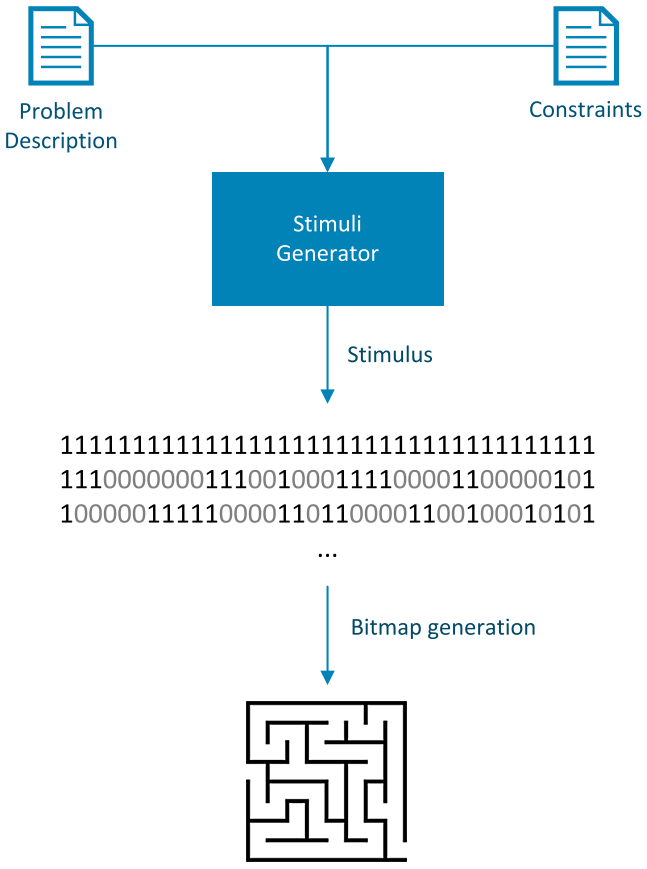


Fig. 21. An idea of generating a maze for the robot controller.

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.

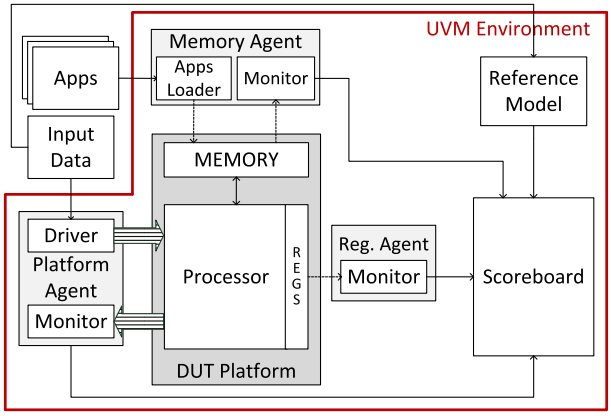


Fig. 22. The UVM-based verification environment for the processor.

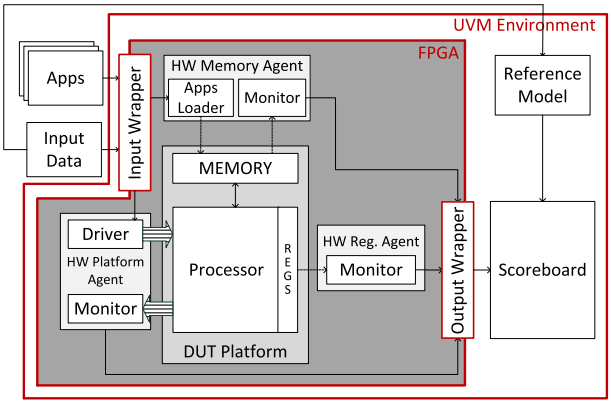


Fig. 23. The architecture of the FPGA-based verification environment for the processor.

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:

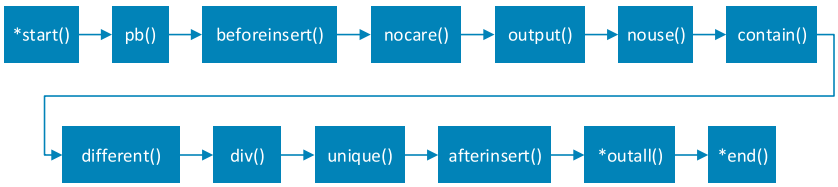


Fig. 24. The set of the constraints for generating the assembly code.

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 src1, imm"}
```

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

The *substitute part* defines the set of strings to be replaced in the instruction syntax. This part is typical for the register definition. 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

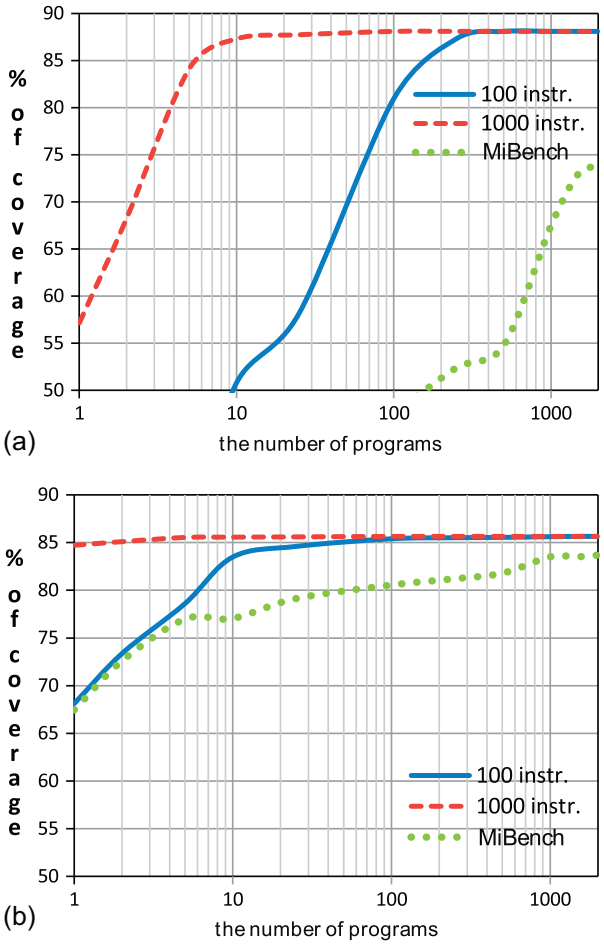


Fig. 25. Achieved (a) instruction coverage and (b) statement coverage in functional verification.

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.

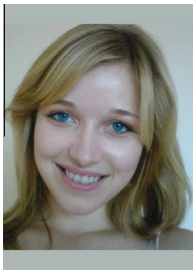
References

[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=qaliX3hYWL4C>.
[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).