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

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

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

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

I. Introduction

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

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

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

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

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

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

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

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

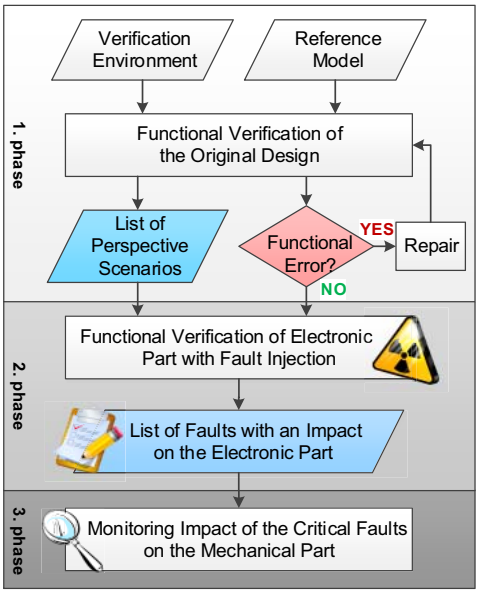

Fig. 1. The flow of phases in the digital systems verification.

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

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

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

II. The First Phase - Verification Environment Architecture

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

A. Verification Environment for Single Verification Scenario

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

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

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

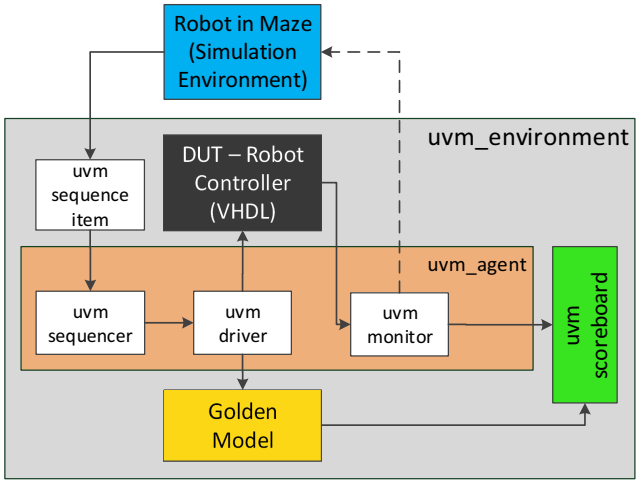

Fig. 2. Verification environment for single verification scenario.

B. Extended Multiple Verification Scenarios Evaluation

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

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

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

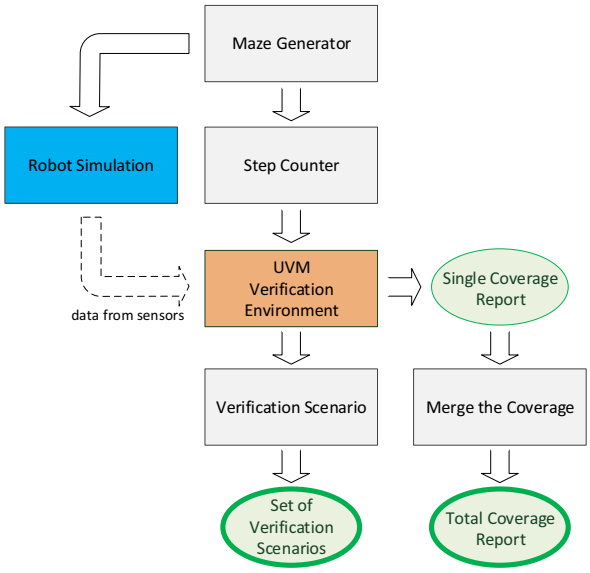

Fig. 3. Extension of verification environment for multiple evaluation.

III. The Second Phase - Evaluation Platform Architecture

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

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

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

A. Architecture of FPGA-based verification environment

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

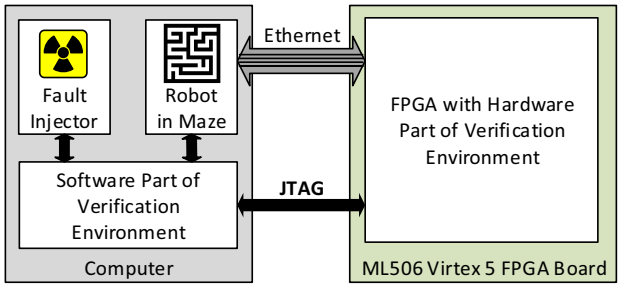

Fig. 4. The structure of the experimental platform.

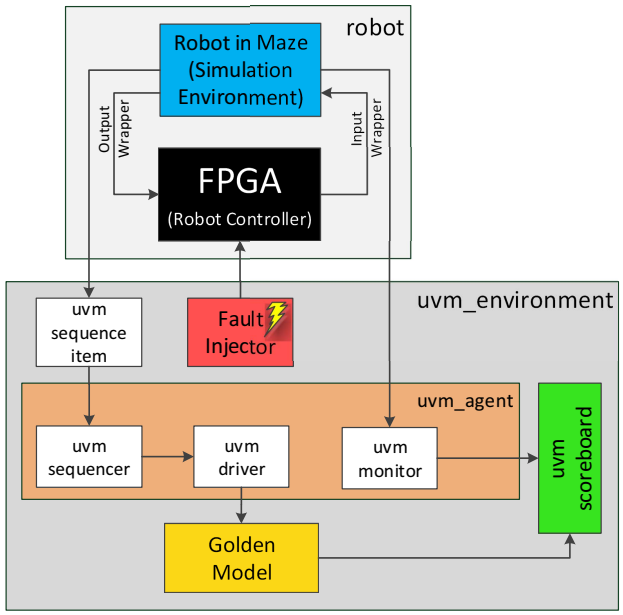

Fig. 5. The architecture of the FPGA-based verification environment.

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

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

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

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

The next step is launching the robot in a simulation and verification environment which provides enable signals to the simulation environment when it is ready to start monitoring. Then, the robot starts to search for paths through the maze which is the proper time for fault injection. It should be noted that fault injection proceeds according to the selected strategy. Our fault injector allows us to inject faults into specified functional units which can be advantageously used. For example, we can inject single faults during one verification run into the specified functional unit, multiple faults into the specified functional unit or inject multiple faults into multiple functional units. After fault injection, the verification run is finished or timeout is expired and then results of the verification are recorded into the verification report.

B. Communication Between Software and Hardware Part

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

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

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

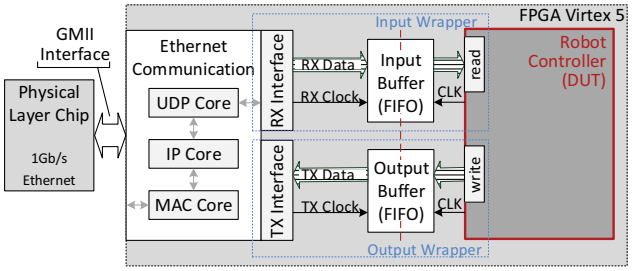

Fig. 6. The architecture of communication between SW and HW part.

C. Evaluation of Reliability by Fault Injection

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

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

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

IV. Maze Generation

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

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

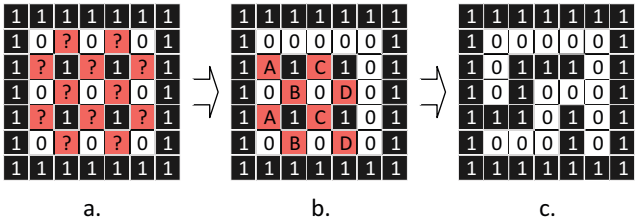

Fig. 7. The demonstration of a conversion of the basic matrix of the maze for needs of the generator.

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

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

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

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

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

----- Constraints -----

constraints {
    nlines(7,7)

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

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

    useonly(odd)
    afterinsert(odd,even)

    end("1111111")
}
```

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

V. Experiments and Results

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

A. The First Phase

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

TABLE I. Average number of robot steps

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

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

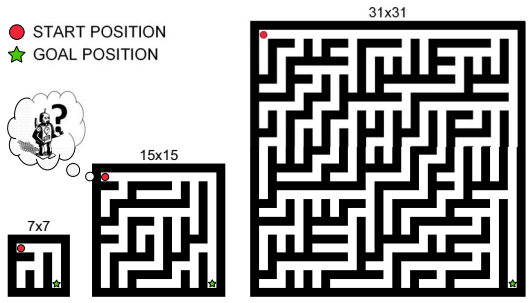

Fig. 8. Three types of mazes.

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

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

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

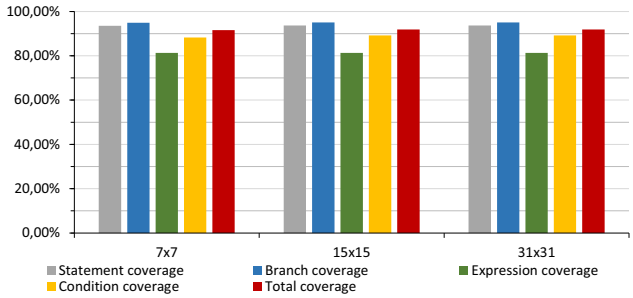

Fig. 9. Code coverage for each type of mazes for 100 verification scenarios.

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

B. The Second Phase

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

TABLE II. THE EXPERIMENTAL RESULTS

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

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

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

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

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

TABLE III. EXPERIMENTAL RESULTS IN FUNCTIONAL VERIFICATION.

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

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

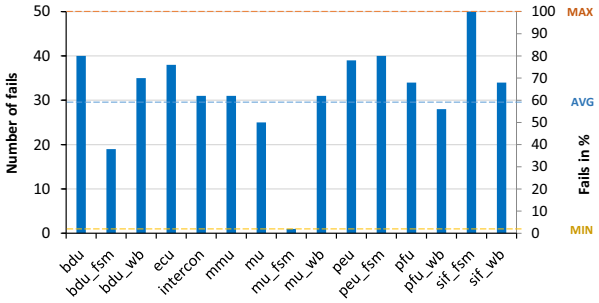

Fig. 10. Experimental results in functional verification.

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

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

VI. CONCLUSIONS AND FUTURE RESEARCH

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

TABLE IV. EXTENDED EXPERIMENTAL RESULTS.

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

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

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

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

ACKNOWLEDGMENT

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science - LQ1602, ARTEMIS JU under grant agreement no 641439 (ALMARVI) and BUT project FIT-S-14-2297.

REFERENCES

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

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

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

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

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

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

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

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

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

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

[11] J. Podivinsky, O. Cekan, M. Simkova, and Z. Kotasek, "The Evaluation Platform for Testing Fault-Tolerance Methodologies in Electro-Mechanical Applications," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*. IEEE, 2014, pp. 312–319.

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

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

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

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

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

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

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

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

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

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