Annales UMCS Informatica AI 4 (2006) 123-134

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Analysing system susceptibility to faults with simulation tools

Piotr Gawkowski, Janusz Sosnowski*

Institute of Computer Science, Warsaw University of Technology,
Nowowiejska 15/19, 00-665 Warszawa, Poland

Abstract

In the paper we present original fault simulation tools developed in our Institute. These tools are targeted at system dependability evaluation. They provide mechanisms for detailed and aggregated fault effect analysis. Based on our experience with testing various software applications we outline the most important problems and discuss a sample of simulation results.

1. Introduction

Digital systems are widely used in various application areas including those with high reliability, availability and safety requirements (e.g. telecommunication, medicine, aviation, industrial control systems, banking). These requirements are the most important in so called dependable systems. To increase system dependability we use in general three techniques: fault avoidance, fault masking and fault tolerance [1,2]. The main idea of fault avoidance techniques is to prevent fault occurrence. This is achieved by design reviews and automation, part selection, screening, lowering power consumption, software rejuvenation etc. Fault masking techniques hide the faults and prevent occurrence of errors using error correction codes or passive redundancy e.g. triple modular redundancy with voting. Fault tolerance techniques detect faults, identify them and perform appropriate recovery (e.g. replacing a faulty model by a spare one).

An important practical issue is to evaluate system dependability. This can be done using various analytical models, collecting reports on system operation from the field and by simulating faults and observing system susceptibility to these faults. This last approach gains high interest in recent years. Various tools have been developed for this purpose [3-10]. They are targeted for models or working systems (execution-based). Simulating faults in a system model (e.g. based on hardware description language VHDL or Verilog) assures high

*Correponding author: *e-mail address*: jss@ii.pw.edu.pl

flexibility, however, it is cumbersome for complex systems. Hence, various fault injection techniques into the working systems are widely reported in the literature [3-5,10,11]. In our Institute we have developed efficient tools for injecting faults into computer systems. These tools are systematically improved and used successfully in our research as well as in didactic. They comprise some original features not encountered in other tools and are recognized by other European institutions involved in dependability problems. In particular, our tools assure better experiment controllability, capability of detailed tracing of fault effects and correlation with various application properties. This has created new possibilities in dependability analysis. In section 2 we describe our fault injectors and give some illustrative experimental results as well as our remarks gained due to research experience with using these tools in the context of other tools. Due to some encountered problems we have developed other specialized supplementary simulation tools described in section 3. In the conclusion we summarize our experience and outline directions of our future research in this topic.

2. Software implemented fault injectors

In 1998 we have developed the fault injector (FITS) operating in Windows environment on IBM PC compatible platform. It has been systematically enhanced and modified. It was also the basis for other simulators targeted for multithreaded applications (MTI) and Linux environment (LIN). Using theses fault injectors for many years in student projects as well as in research, we have gained rich experience. The latest version of FITS proved that it is the one of the most complex, flexible and easy to use SWIFI (software implemented fault injector) tool reported in literature. It uses standard Win32 Debugging API to control the execution of the software application under test. During the so called Golden Run (GR) the execution trace as well as the execution results are saved in a log file (GRL). Additionally, statistic information is gathered on the tested application e.g. resource usage, code size, instruction distribution. FITS, as opposed to most of other SWIFI tools reported in literature, works on a single IBM PC compatible computer under Win32 operating systems family (Windows NT, 2000, XP). The experiments can be done on the executable code of the application (this is important if the source code is not available). However, some source code modifications (described later) can be helpful to simplify fault effect propagation analysis.

In FITS faults are simulated by disturbing the running application. In this process an important issue is the type, location and time of fault injection. To assure better experiment controllability we admit so called *testing areas*. Such areas can be marked with the use of the predefined *magic sequence* in the source code of the application (every odd occurrence of this sequence begins new testing area while every even – closes current testing area). There can be many

testing areas defined within a single execution of the application. Another possibility to mark the testing area is to define two addresses of the instructions in the application code – execution of the first one starts testing area, execution of the second one – terminates it. Here no source code modifications are needed. Testing areas are very useful. They can limit the scope of disturbances only to the most interesting parts of the analyzed application. Additionally, they allow creating some extra code, not disturbed during experiments, sometimes necessary to run the analyzed application.

Another optional modification of the application to be tested is the insertion of *user-messages*, which are captured and collected by the FITS during experiments. This mechanism provides supplementary communication between the tested application and FITS and has no impact on the application behavior as the communication channel used (Win32 Debugging API) guarantees that. Using this mechanism, the tested application can signal detected errors, the path of fault propagation, efficiency of fault forecast boundaries etc. This simplifies tracing fault effects.

An important issue in experiments with fault injectors is qualification of results. This problem was neglected in literature due to the fact that the authors in most cases analyzed simple calculation-oriented applications where the incorrect results are easily identifiable. It is much more complicated for other classes of applications e.g. oriented at database, document or signal processing, real-time applications in process control. First of all, the result of such applications has to be identified taking into account its aspects related to value, time of delivery, impact on performed data processing etc. In some applications the accepted levels of result deviations as well as error severity should also be defined. For example, in controlling processes of a mechanical object, spurious temporary fault control signals can be tolerated by the object inertia. To resolve this problem, in FITS the correctness oracle is specified in accordance with specific features and characteristics of the tested application. Some illustration is given in [10,12,13].

2.1. Experiment setup

FITS can emulate permanent errors as well as Single or Multiple Event Upsets (SEU and MEU) related to transient faults, which dominate in contemporary systems [1,14,15]. Faults can be injected into the main resources available at the machine code level of the application. A fault to be injected is defined by the type of modifying operation (e.g. XOR, SET, RESET) performed on the target *fault location* (e.g. a register) in specified (in a so called bit-mask) bit positions. The experiment is configured using convenient GUI interface (an illustrative window is given in Fig. 1).

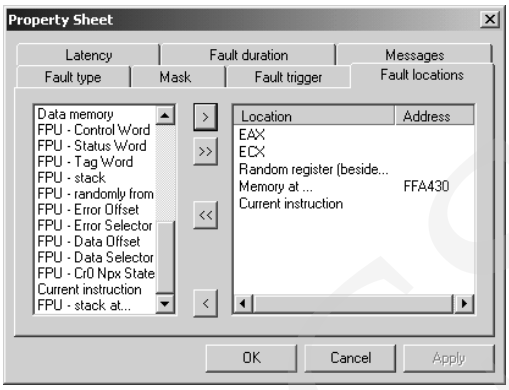

Fig. 1. FITS window for defining fault injections

The duration of a faulty state can also be programmed as a period from one machine cycle to even permanent faulty state. In FITS it is also possible to mimic the effects of complex faulty behavior or source-code level errors (e.g. execution of additional user-defined code, delays in processing paths, coding errors).

The fault injection process is preceded by execution of the analyzed application (without faults) in the so-called Golden Run mode. In this mode FITS traces the application execution and collects various statistics. The collected information helps to profile the experiments (e.g. by showing *Activity Ratio* [12,13,16] of resources), in particular facilitates fault distribution over execution time and resource space [13,16,17]. To achieve higher efficiency, the target location is disturbed just before being used (by the application under test). Faults can be located in CPU registers, application code, stack, data memory, FPU etc. Fig. 1 shows FITS dialog window for selection of fault locations. The list on the left side contains all possible fault locations. The user can select any of them and move with the buttons to the list of selected locations (list on the right side of the dialog). Faults can be defined explicitly by the operator or generated pseudorandomly. FITS assures the experiment repetitiveness, which is useful in deeper analysis of fault effects. This is a unique property, not available in similar tools reported in literature. Injected faults can be emulated for the explicitly defined set of faults (with specified location and triggering moments) or generated automatically with a specified profile.

2.2. Experiment reports

A fault injection experiment is composed of tests specified in some preset configuration. Each test relates to a single execution of the tested application with an injected fault (or faults). All side effects of injected faults are stored, so the fault propagation can be traced in detail. However, some aggregation and

filtering functions (to limit the volume of the collected data) can also be included. An example of a single fault injection test report is shown in Fig. 2.

```
Address of instruction at which fault was injected: 401297
Instruction execution moment: 14
Fault location: RANDOM within instruction
Instruction before first injection: 00401297:  77c9        ja     00401262
Instruction after first injection:    00401297:  37          aaa
Fault mask: randomly selected one bit
Bit disturbance operation: XOR
Fault duration: 1 instruction
Program was terminated
Exit code:254 was not correct
Messages during execution:
* Access Violation while reading - first chance at Eip=4012A3h
* Access Violation while reading - first chance at Eip=10208h
* Access Violation while reading not handled by debugee at Eip=10208h
  - Second time - terminating
```

Fig. 2. A sample of a single test report

It comprises fault injection time (instruction address and its execution moment – 14th), location etc. This fault (single bit flip in instruction code) resulted in changing the instruction to be executed from conditional jump (*ja 0x00401262*) to arithmetic adjust (*aaa*). That led the application to two exceptional situations (listed as messages). The first one was generated by the attempt to read unavailable memory by instruction at the address 0x004012A3. FITS gives the possibility to handle the first occurrence of exception by the application (specified as the *first chance*). Unfortunately, the injected fault generated another (second) exception – this time not handled, so at the second occurrence of the same exception the application was terminated.

For experiments with many fault injections (e. g. generated pseudorandomely in a specified resource) all tests are performed automatically and beyond detailed test reports we obtain aggregated results. In particular, we can get the percentage of results registered within specified 5 categories: C – correct result, INC – incorrect result or wrong timing of its delivery, S – fault detected by the system (specification of the number and types of registered exceptions), T – time-out, U – user-defined messages generated by the tested application (e.g. signalling faulty behaviour). Moreover, in aggregated experiment report all user-defined messages are listed with the number of its appearance. There is also a possibility to deliver experimental results to the database for easier further processing and visualization. In embedded systems the correct result classification may admit some minor anomalies e.g. short output signal transients, delays etc. Most

embedded applications run continuously but the result observation cannot be infinite process. We can limit it to some delay after the fault injection and control some internal states, which may have influence in the future operation.

2.3. Examples of simulation results

The usefulness of FITS was verified in many student projects and research studies. To give a better view on experimentation capabilities we give a sample of illustrative results for a calculation-oriented application. The analyzed program is the Fast Fourier's Transformation (FFT) from publicly available library – FFTW [13]. We have analyzed two versions of this program:

- V1 – the original simple implementation of FFT calculation, with neither fault detection nor fault tolerance mechanisms,
- V2 – modified application V1 comprising original fault detection and fault tolerance mechanisms introduced in [13].

Version V2 is based on time and code redundancy enhanced with exception handling. Input data is processed in iterations. Each iteration is processed twice by replicated COTS components (code and time redundancy). Moreover, it is guarded by the exception handling mechanism. This protects (with high coverage) the controlling algorithm from possible disturbances caused by a faulty processing component. The obtained pair of results is compared and in case of consensus the result is delivered as the application output. Any detected exception or disagreement between results initiates internal testing and recovery procedure (data and code consistency and recovery). It is followed by the third repetition of calculations and then voting over three results. Implementation details are given in [13].

Both versions V1 and V2 were tested with FITS and disturbed by single bit-flip faults injected randomly into the application code (instructions), CPU registers, FPU, application stack and data memory area. Around 1000 tests were made for each fault location. Experimental results are presented in Fig. 3. It shows the percentage of basic test categories (C – correct, INC – incorrect, S – system exceptions, T – time-outs) depending upon the location of injected faults (instructions, registers etc.). For each location we give a pair of bars. The left one corresponds to the V1 version and the right one to the V2 version.

The experiments proved high efficiency of implemented fault tolerance mechanisms for the considered fault model. We have observed that some exceptions are not intercepted at the application level (to perform correction), so the application is terminated by the operating system. This results from some discrepancies of Windows, hence, further research targeted at improving this issue is required. Table 1 gives a representative distribution of not handled exceptions in Win32 environment as a consequence of single bit-flip faults in the instruction codes.

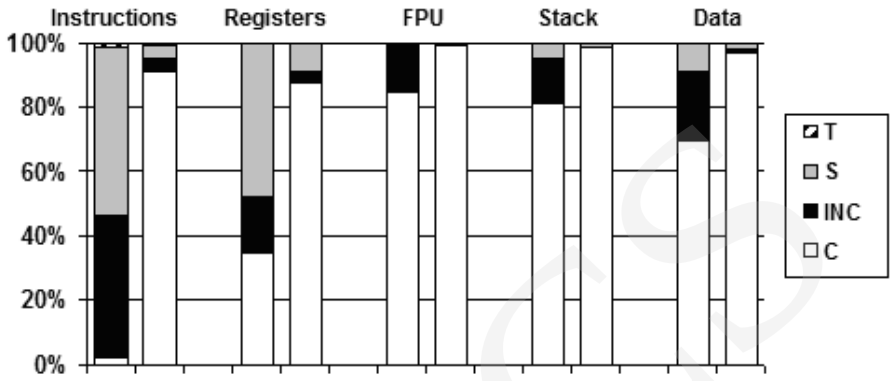

Fig. 3. Fault susceptibility of two versions of FFT algorithm

Table 1. Distribution of exceptions

Exception	%
ACCESS VIOLATION	80.6
ILLEGAL INSTRUCTION	5.5
STACK OVERFLOW	4.8
BREAKPOINT	4.7
PRIVELAGE INSTRUCTION	4.3
ARRAY BOUNDS EXCEEDED	0.1
INTEGER OVERFLOW	0.1

In the case of faults injected in the application code the most effective system mechanism of fault detection is Access Violation. The memory management unit triggers it, if the application violates memory access rights. Significant percentage of access violation results from the high probability of disturbing referenced addresses in such a way that unreachable memory regions are targeted. Hence, the classical control flow checking mechanisms (e.g. [1,18,19]) are not so effective in the Win32 applications as in embedded systems, which usually do not comprise memory protection mechanisms.

In experiments with various applications we observed different fault susceptibility in different segments or modules of these applications. Moreover, the probability of fault occurrence in a memory is a function of the used memory space. Sometimes parts of the application with high fault sensitivity are executed scarcely but occupy a large memory space. So, the distribution (in time and space) of injected faults has to be programmed carefully during the experiment setup. These issues are analyzed in [13,16,17].

3. Supplementary simulation tools

To evaluate system dependability with fault injectors we have to simulate a large number of faults in order to assure statistically significant results. Moreover, to obtain representative results it is important to assure compatibility of the test profile with the operational one [4,16,20]. For this purpose, we can use statistics of input data or module utilisation and select representative test scenarios to cover all possible situations. In this process some coverage measures are helpful. We can deal with functional or structural coverage of the application. Functional coverage is related to the application specifications. Structural coverage can be related to program data and control flow [16]. We use a special tool, which measures such structural features as:

- *block coverage* – coverage of blocks composed of code fragments without branching (program is composed of branch free segments of code comprising entry and exit points),
- *decision coverage* – measures the portion of decisions executed during testing,
- *c-use coverage* – counts the number of combinations of an assignment to a variable and a use of the variable in a computation that is not part of conditional expression,
- *p-use coverage* – counts the number of combinations of an assignment to a variable, a use of the variable in a conditional expression and all branches based on the value of the conditional expressions,
- *all-use coverage* – c -use or p-use,
- *du-path coverage* – counts the number of paths from a variable's definition to its use, which contains no redefinition of the variable,
- *path coverage* – coverage of all allowed sequences of statements in the program (practically not used).

SWIFI injectors are universal tools with high flexibility in the area of experiment controllability and observability. This universality may create some problems in specialised fault injection strategies related to time overhead of injections as well as the necessity of complex specification of fault triggering. Hence, for some well-defined problems it is more reasonable to use dedicated fault simulators. In particular checking fault susceptibility of complex data structures used by many applications is more efficient with specialised simulators. Such complex data structures are used in applications dealing with documents and databases etc. These structures comprise information related to the processed data (e.g. text codes, graphical objects) and various control information supporting performed operations. Moreover, some redundancy or error detection and correction features are included. In contrast to calculation-oriented applications, complex data structures show higher fault robustness. In particular a large percent of faults has no influence on the operation due to some redundancy. On the other hand, these applications perform various checking

functions so many faults generate messages signalling incorrect operation. The list of these messages is quite long. We have also analysed the fault susceptibility distribution in function of fault location within the tested documents. Some areas storing control information generated most of messages.

In the specialized simulators we can adapt fault injection locations in correlation with logical structure of the analyzed data structures. On the other hand, fault propagation effects are also controlled in a more efficient and application oriented way. Here we may have specific signalling of errors by the application (error messages), identification of abnormal states (e.g. hanging, performing partially specified functions) etc. In specialized simulators we can define test scenarios in the form of a sequence of appropriate actions with appropriate data (simulation of an operator) specific for the considered class of applications. In the case of calculation-oriented applications test scenarios can be expressed only as simple categories of appropriate data sets.

In many applications, while qualifying their test results, we can admit some disturbances as acceptable e.g. in text, graphical or sound files data disturbances may reduce the quality of the represented information still preserving the logical significance (text, image or sound are recognisable). Hence, while we evaluate the test result we have to use application dependent qualification procedures. We will illustrate this for a sample of results with randomly disturbed (bit-flip errors) files by means of special simulators. These results relate to MS Excel, LATEX, graphical and multimedia sound files.

While analyzing MS Office document fault susceptibility we have injected single bit upsets randomly within the document area (equal distribution in space). For MS Excel the injected faults generated only 12.6% of errors. An interesting thing is that different document areas have various susceptibility to faults. Some are either not used or redundant so no error was reported. Detected errors by MS Excel related to mechanisms checking document format and context of control information. Quite interesting is the distribution of document errors: 17% have not been detected by the program (INC), 46% faults generated Excel messages informing on the impossibility of loading document, 20% faults were signaled as access violation, 12% faults were not visible after opening document, however, new writing of the document generated Excel warning messages or exceptions. In 5% of cases error messages (e.g. lack of free memory) were not consistent with real faults.

LATEX files comprise text, commands, mathematical equations, comments etc. Depending upon fault rate we could use the file or not (if some important structural data is disturbed). For fault rates lower than 1/128 in 90% of cases the file was available for processing. However, legible documents (with acceptable errors) appear for fault rates at the level of 1/1000. It is interesting that disturbing mathematical formulas with the fault rate up to 1/64 were acceptable.

In graphical files (e.g. JPG, BMP) an important issue is the level of image disturbance. JPG files are accessible in over 90% for fault rates below 0.005 with possibility of general recognition of the image (disturbed in about 30%). BMP files are very susceptible to faults in the area of the header (even single upsets may block access to the file). Faults in data area lower image quality. For the fault rates below 0.001 practically these disturbances are not visible.

Information structure *wav* for storing and processing sound comprises a header (header length, number of sound channels, coding technique, average sample rate etc.) and sound data samples. Disturbing *wav* files with transient faults we observed that for the fault rates higher than 1/40 over 90% of files were accessible but the play function in most cases was rejected. For the fault rate 1/128 this function practically was available. The header area is very sensitive to faults (even single upsets are critical). Faults in data part only degrade sound quality, for the fault rate 1/10 the noise is very high but sound is recognizable. We have also analyzed the impact of packet losses during transmission of *wav* files via Internet. For transmitted music listeners did not observe quality degradation (subjective evaluation) up to 3% of packet losses. Losses up to 10% were acceptable. For 30% losses sound quality was very bad, nevertheless still recognizable.

4. Conclusions

The developed fault simulation tools have been successfully used by many M.Sc and Ph.D students in projects, diploma thesis and our research works. The obtained results attracted the attention from other institutions active in dependability research. In particular we compared simulation results obtained with the model based fault injector MODELSIM [21] developed in Grenoble TIMA3 laboratory for the car immobiliser developed in our Institute. Our execution based fault injector assures much higher speed of fault injections.

SWIFI injectors assure relatively high speed. However, they have some limitations in simulating faults at the level not accessible to the programmer. Hence, in practice it is reasonable to develop hierarchical simulation. For example characterising faulty behaviour of a module by simulating faults at a lower level (e. g. electrical or RTL model). Such characterising experiments are also reported in literature (e. g. [22-24]). Another critical problem relates to experiment result qualification. This problem is not trivial in real time applications, where some time or result value deviations can be considered as acceptable as well as short transient pulses (tolerated by the inertia of the controlled system). This problem is correlated with selecting representative test scenario.

Recently many fault tolerance techniques in software have been proposed (e.g. [13,19,25-30]). They are especially efficient for transient faults dominating in contemporary systems [1,14,15]. The proposed ideas in literature can be

implemented in different ways, so their optimization is needed. In this process fault injectors are indispensable. Moreover, they are useful in dependability evaluation of existing solutions.

In our further research we concentrate on increasing fault injection effectiveness by distributing the fault injection processes in a computer network. Other improvements can be achieved by extracting system behaviour in different operation time frames and faster qualification of the experiment results (e. g. by observing selected internal state variables). We also develop a new fault injector for microcontroller systems. It is based on an original RTL level simulator [31].

Acknowledgment

This work was supported by KBN grant 4T11C049 25.

References

[1] Sosnowski J., *Transient fault tolerance in digital systems*. IEEE Micro, (1994) 24.
[2] Sosnowski J., *Testowanie i niezawodność systemów komputerowych,* EXIT, (2005), in Polish.
[3] Arlat J., et al., *Comparison of physical and software implemented fault injection technique*. IEEE Trans. on Computers, **52**(8) (2003) 115.
[4] Benso A., Prinetto P., *Fault injection techniques and tools for embedded systems reliability evaluation,* Kluwer Academic Publishers, (2003).
[5] Carreira J., Madeira H., Silva J.G., *Xception: a technique of the experimental evaluation of dependability in modern computers.* IEEE Trans. on Software Engineering, 24(2) (1998) 125.
[6] Chen M., Tsai T.K., Iyer R.K., *Fault injections and tools.* IEEE Computer, (1997) 75.
[7] Civera P., et al., *Exploiting FPGA based techniques for fault injection campaigns on VLSI circuits*. Proc. of IEEE DFT Symposium, (2002) 250.
[8] Leveugle R., *Fault injection in VHDL description and emulation.* Proc. of IEEE DFT Symp., (2000) 414.
[9] Samson J.R., Moreno W., Falquez F., *A technique for automated validation of fault tolerant designs using laser fault injection.* Proc. of IEEE FTCS-28 Symp., (1998) 162.
[10] Sosnowski J., Gawkowski P., Lesiak A., *Software implemented fault inserters.* Proc. of IFAC PDS2003 Workshop, Pergamon, (2003) 293.
[11] Carderilli G.C., Kaddur F., Leanori A., Ottavi M., Pontarelli S., Velzaco R., *Bit flip injection in processor based architectures: a case study*. Proc. of 6th IEEE On-Line Testing Workshop, (2002) 117.
[12] Gawkowski P., Sosnowski J., *Dependability evaluation with fault injection experiments.* IEICE Trans. Inf. & Syst., E86-D(12) (2002) 2642.
[13] Gawkowski P., *Analysing and enhancing fault immunity of programs in systems with COTS elements.* Ph.D. thesis, Institute of Computer Science, Warsaw University of Technology, (2005).
[14] Dodd P.E., Massengill L.W., *Basic mechanisms and modeling of single-event upset in digital microelectronics*. IEEE Trans. on Nuclear Science, 49 (2003) 583.
[15] Normand E., *Single Event Upset at Ground Level,* IEEE Trans. Nucl. Sci., 43 (1996) 2742.
[16] Sosnowski J., Gawkowski P., Lesiak A., *Fault injection stress strategies in dependability analysis. Control and Cybernetics*, 33(4) (2004) 679.
[17] Gawkowski P., Sosnowski J., Radko B., *Analyzing the effectiveness of fault hardening procedures.* Proc. of the 11th IEEE Int'l On-Line Testing Symp., (2005) 14.
[18] Benso A., Di Carlo S., Di Nartale G., Prrinetto P., Tagliaferri L., *Control flow checking via regular expressions*. Proc. of the 10th Asian Test Symposium, (2001) 299.

[19] Oh N., Shrivani P.P., McCluskey E.J., *Control flow checking by software signature*. IEEE Trans. on Reliability, 51(1) (2002) 111.
[20] Madeira H., Some R.R., Costa F.D., Rennels D., *Experimental evaluation of a COTS system for space applications.* Proc. of IEEE Int. Conf. on Dependable Systems and Networks, (2002) 325.
[21] Velazco R., Mochnacs J., Peronnard P., Calvo O., *Analysis of the criticality of transient bit-flip faults in a massive embedded application.* Proc. of IEEE Latin America Test Workshop, (2004) 178.
[22] Kim S., Somani A.K., *Soft Error Sensitivity characterisation for Microprocessor Dependability Enhancement Strategy.* Proc. of IEEE Int. Conference on Dependable Systems and Networks, (2002) 416.
[23] Pleskacz W., Kasprowicz D., Oleszczak T., Kuźmicz W., *CMOS standard cells characterization for defect based testing.* Proc. of IEEE DFT Symp., (2001) 384.
[24] Saggese G.P., Vetteth A., Kalbarczyk Z., Iyer R., *Microprocessor Sensitivity to Failures: Control vs Execution and Combinational vs Sequential Logic.* Proc. Of Int. Conf. on Dependable Systems and Networks, (2005) 760.
[25] Cheynet P., et al., *Experimentally evaluating an automatic approach for generating safety critical software with respect to transient errors.* IEEE Transactions on Nuclear Science, 47(6) (2000) 231.
[26] Nicolescu B., Velazco R., Reorda M.S., *Effectiveness and limitations of various software techniques for soft error detection, a comparative study.* Proc. of 7th IEEE Int. Testing Workshop, (2001) 172.
[27] Oh N., Mitra S., McCluskey E .J., *ED4I Error detection by diverse data and duplicated instructions.* IEEE Trans. on Computers, 51(2) (2002) 180.
[28] Oh N., Shrivani P.P., McCluskey E.J., *Error detection by duplicated instructions in super scalar processors*. IEEE Trans. on Reliability, 51(1) (2002) 63.
[29] Rebaudengo M., Reorda M.S., Violante M., *A new software-based technique for low cost fault-tolerant application.* Proc. of Annual Reliability and Maintainability Symp., (2003) 25.
[30] Vargas F., et al., *Briefing a new approach to improve the EMI immunity of DSP systems.* Proc. of IEEE Asian Test Symposium, (2003) 468.
[31] Wilczyński A., Sosnowski J., Gawkowski P., *Flexible microcontroller simulation for testing purposes.* Proc. of IFAC Workshop on Programmable Devices and Systems, (2004) 310.