INFORMATYKA
CZASOPISMO TECHNICZNE
TECHNICAL TRANSACTION
COMPUTER SCIENCE AND INFORMATION SYSTEMS
WYDAWNICTWO POLITECHNIKI KRAKOWSKIEJ
1-I/2011
ZESZYT 9
ROK 108
ISSUE 9
YEAR 108

DAWID TRAWCZYŃSKI, JANUSZ SOSNOWSKI*

FAULT INJECTION TESTING OF SAFETY-CRITICAL APPLICATIONS

TESTOWANIE APLIKACJI KRYTYCZNYCH ZA POMOCĄ WSTRZYKIWANIA BŁĘDÓW

Abstract

In the paper, we discuss an original methodology of dependability evaluation dedicated for safety-critical embedded systems. It is based on a fault simulation technique known as Software Implemented Fault Injection (SWIFI). This methodology combines functional and structural models to achieve higher modeling accuracy than existing approaches. The method was implemented and verified on a representative safety-critical case study, namely the automotive anti-lock braking system.

Keywords: software implemented fault injection, real-time embedded systems, dependability analysis, testing

Streszczenie

W artykule przedstawiono oryginalną metodologię oceny wiarygodności krytycznych aplikacji wbudowanych opartą o tzw. programową symulację błędów (SWIFI). Metodologia ta łączy modele funkcjonalne i strukturalne w celu dokładniejszej (niż dotychczasowe podejścia) analizy wrażliwości na błędy w systemach wbudowanych. Została ona zaimplementowana i zweryfikowana na reprezentatywnym przykładzie systemu czasu-rzeczywistego, sterującym hamulcami samochodowymi (algorytm przeciwpoślizgowy ABS).

Słowa kluczowe: programowe wstrzykiwanie błędów, wbudowane systemy czasu-rzeczywistego, analiza wiarygodności, testowanie

* Dr inż. Dawid Trawczyński, prof. dr hab. inż. Janusz Sosnowski, Instytut Informatyki, Wydział Elektroniki i Technik Informacyjnych, Politechnika Warszawska.

Paper submitted for publication: 22.04.2010

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

1. Introduction

Recently, safety-critical control systems often use complex electronics to realize their functionality. An important issue is to assure their dependable, reliable and secure operation to prevent safety hazards. To provide these properties, a system must be able to detect and correct various faults that include: environment disturbances, design faults, interaction faults, aging faults, etc. Detection and correction of these faults is a challenging problem and requires scalable and adaptable testing techniques (high fault coverage). This can be achieved with the experimental technique called Software Implemented Fault Injection (SWIFI) introduced in 90s [11, 12, 14, 15], and extensively developed recently ([1, 2, 19, 22, 30, 31] and references therein). The main objective of SWIFI approach is to artificially accelerate the occurrence of faults and efficiently analyze system's behavior in their presence. Such analysis gives a good insight into the behavior of a system disturbed by faults. SWIFI provides the capability of simulating a large number of fault conditions during system operation that can affect such resources as CPU registers, ALU, FPU, cache memory, data bus, I/O interfaces, etc. Moreover, SWIFI method can emulate many real faults occurring due to environment disturbances (electromagnetic, radiation interference, etc.) that affect contemporary integrated circuits. Such emulation can be done in a well controlled manner, often not possible in other approaches. Finally, SWIFI testing can provide data for analytical reliability analysis (e.g., Markov modeling).

SWIFI testing of control systems presents new challenges usually not encountered in calculation-oriented applications (mostly discussed in the literature). The control programs may comprise fewer data-oriented computations, more of control-flow expressions and interactions with the environment. Hence test scenarios usually are more sophisticated and often result in a number of acceptable behavior trajectories for which some deviation is acceptable. So test qualification (assessment of system behavior) is much more complex than in calculation oriented applications. Analyzing control systems we have to assure representative controller, environment, and controlled object models, together with test scenarios and vectors. This allows emulating essential behavior of real systems during faulty and fault-free conditions taking into account feedback and environment inertial properties. This in practice may involve some natural fault masking and "self-correction" effects. Performing SWIFI tests, it can be important to assure code separation between various system parts (e.g. microcontroller and environment models) to assure good experimental controllability and avoid unexpected fault propagation within the system. An important issue is to assure flexibility in dealing with various versions of models, control programs, etc. This is useful in checking fault hardening mechanisms. Developing a new SWIFI test methodology targeted at control applications, we took into account the above listed issues. Moreover, we have integrated various simulation techniques. Such large scope of problems is not covered in other publications.

The paper is organized as follows. Section 2 discusses related work. Section 3 presents our simulation methodology based on SWIFI. Section 4 explains the considered case-study model (car ABS system) that was used to illustrate and verify our methodology. In section 5, we discuss experimental results related to several implementations of the ABS system. Final conclusions and an outline of future research are given in section 6.

BPK BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

2. Related work

A few dependability simulation studies perform a complete evaluation of real-time systems. Principal difficulty in studying such systems is the lack of standard fault models, and significant time and effort required to develop controller and environment models. Moreover, one usually requires knowledge from multiple domains (real-time systems, embedded programming, and control systems) to implement necessary simulations. Interesting approaches to real-time system dependability evaluation via simulation are presented in [4, 5, 8, 9, 18] and [27].

Papers [8] and [9] deal with the model-based dependability simulation technique. Here the authors evaluated, with SWIFI, the fault susceptibility of a Model Predictive Control (MPC) control algorithms supporting the alcohol rectification process and controlling a robotic arm. The main conclusion is that the qualification of the real-time systems response in the presence of controller faults is significantly more complex than in calculations-oriented applications because it must consider time-dependant system trajectories for which some deviation is acceptable. Moreover, the authors show that fault susceptibility of the modeled MPC algorithm, can be reduced by using structured (Win32) exception handling. The simulations processes were restricted only to the considered control algorithm.

In [4, 5, 18, 27], and [17] authors discuss various real-time systems (i.e., traffic light controller, jet engine controller, inverted pendulum, hot-air blower) that were tested using SWIFI tools. Moreover, in [16] the author focuses attention on specific fault tolerant protocols implemented on target architecture. All these studies were performed on single processor architectures and concentrated on the analysis of the executed real control algorithms.

We conclude that few works study the dependability (via software fault injection) of a complete, real-time embedded system by modeling and simulation of its most essential components, namely a controller, environment, real-time operating system (kernel) and network. Most works usually address either a simple real-time application, a specific fault tolerant protocol (e.g., TTA/TTP in [13, 16]), or partial model-based simulation in which important components of a real-time system are skipped (e.g. [8]). In [6, 7] only transmission protocols are tested with injected faults.

Many real embedded systems are implemented in multiprocessor environment, use complex software, with communication networks and various physical environments or controlled objects, etc. Hence, arises the problem of integrating so many aspects in one simulation platform (neglected in the literature). To deal with this problem, we developed and combined controller and environment models, and studied the real-time system's dependability with a SWIFI tool integrated with the model of a controller and real-time network functioning within the context of a real-time operating system kernel. This methodology can be used for complete systems as well as in the initial stages of their design, where only functional models are available. In addition to standard fault models (bit flips) considered in the literature, we also address new fault models. Next, we discuss the developed methodology in the context of dependability evaluation and improvement related to real-time, embedded systems (based mostly on software mechanisms).

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

3. Dependability simulation methodology

This section presents the developed simulation methodology that integrates functional and structural models to achieve more realistic representation of system behavior than other approaches. The integration relates to controller, environment, network and fault models used to simulate the behavior of a system in various test scenarios. In our approach, we disturb (using SWIFI) the controller and network model, to assess the behavior of an embedded system under faulty and fault-free conditions. The general model of our system used in SWIFI simulation is shown in Figure 1.

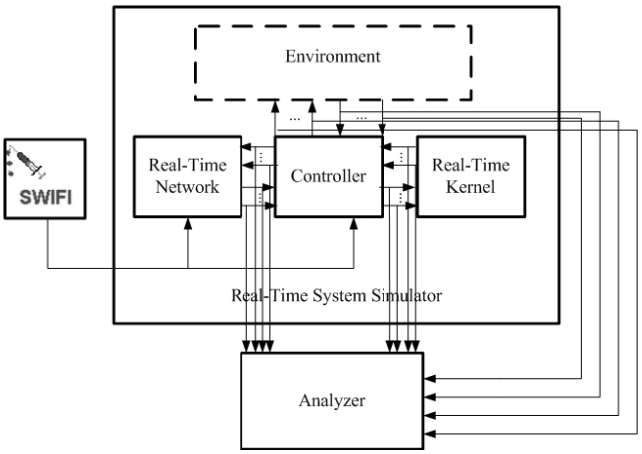

Fig. 1. Components of a real-time system evaluated with SWIFI

Rys. 1. Komponenty systemu czasu-rzeczywistego analizowane poprzez SWIFI

The experiment test bed comprises six main components: SWIFI, environment, controller, network, kernel, and analyzer. Note that in Figure 1 we mix hardware (i.e., the controller) and software components (e.g., the real-time operating system kernel). This is consistent with embedded systems which comprise hardware and software. The operating system kernel actually manages the controller by executing its task schedule, and thus the two can also be considered to be in the same block. However, here we separate the software (kernel) from hardware with the application program (controller) to show that we test the controller, and not the kernel.

The main component of the test bed (Fig. 1) is the controller. It receives inputs and computes control outputs that affect the controlled object, which is represented within the environment. The environment can also comprise objects that are not directly controlled by the controller (e.g., weather). Note, that such objects can also implicitly affect the controller's decision by disturbing the controlled object. The controlled object itself provides a response that is measured by a sensor (e.g., a thermocouple), and in feedback control systems, this measured value is used to generate future controller outputs. In Figure 1, we additionally use the analyzer object that monitors inputs and outputs of the controller, network, and kernel, to assess (via process called *result qualification*) the dependability and performance of the system under test.

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

The real-time kernel provides common services of the operating system such as interrupt handling and scheduling, and task scheduling, activation and suspension. In our experiments, we do not simulate faults occurring in the operating system kernel (this problem is covered in [21]). The real-time network provides timely message exchange among distributed controllers [17]. The fault injector is a tool that disturbs resources used by the controller and the network. It emulates real faults (internal and external).

Our methodology that combines and simulates a controller, environment and fault models consists of the following steps:

– model development and verification,
– system behavioral analysis and testing,
– application instrumentation,
– fault space exploration,
– test scenario development.

The initial step of our dependability evaluation methodology is the design and development of the application functional model. In this step, we combine the controller, and environment models with the real-time kernel and network models, to support a complete, real-time embedded application model. The behavior of this model must then be verified to make sure that it matches expected (i.e., reference) behavior. Moreover we have to analyze its properties and generate representative test cases.

After the model has been verified, we then proceed to adapt the application for software implemented fault injection (SWIFI) testing. Usually this adaptation involves adding fault emulation functions, result qualification program code (checks whether the stressed program executed correctly or not – see the analyzer in Figure 1), and various trap instruction that will be used to break the program's execution at run-time to modify data contents located in processor resources (e.g., memory or register values). Moreover, the adaptation can involve special compilation techniques (e.g., optimization) that generate efficient binary object files used for machine instruction based fault injection. These object files may need to be compiled in a special way as to avoid code sharing between the part of the application that is to be disturbed (e.g., a controller), and the other parts that should remain undisturbed (e.g., the environment).

After the application is verified and adapted to SWIFI testing, a fault injector tool is used to disturb the selected test regions of the application (e.g., the controller program segments). Usually, in this step, thousands or millions of faults are injected e.g. emulating latched (bit flips in memory cells, registers) and non-latched (temporary state changes) transient faults. Moreover, in this step, we also inject more abstract (high-level) faults that disturb the network protocol's MAC FSM, message transmission time, controller task execution time, and the time reference (clock) [23–26]. In each fault injection scenario called simply a test, we inject only one fault and analyze the response of the system. If the response is as expected we qualify it as correct, otherwise we conclude that a fault caused an error and qualify the response as incorrect (other possibilities are given later on). This qualification is performed by the analyzer. We repeat this process for every test that makes up the test suite called here simply as the experiment.

The goal of experiments is to evaluate fault susceptibility of the analyzed systems and identify their weak points. This is helpful to improve their dependability by introducing various fault detection and fault handling mechanisms. For this purpose we use exception handling, FPU resetting, code-reloading, and various watchdog timers. The modified, fault

BPK BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

hardened applications can be further tested with fault injection procedures. This process can be repeated until achieving the required dependability level. When applying our methodology specific issues must be addressed, and we discuss them in the sequel.

3.1. Model development and verification

The controller and environment models are created basing on a functional (behavioral) description. They are verified in Matlab simulations. We verified our results by comparing them with those presented in [20]. Our development approach required generation of various transfer functions (e.g., signal filters) that determined component input response in the time and frequency domain. In case of the environment, we developed models based on differential equations that described its dynamics. We found the environment modeling especially challenging because it can require parameters that are difficult to obtain analytically and must be generated empirically (e.g., tire parameters in car suspension models). Moreover, we assure code separation for the environment and the controller in developed models. This can be done by allocating different memory locations for shared parameters and variables in the controller and environment. Such separation assures that stimulating faults in the controller we do not affect the environment properties.

3.2. System behavior analysis and testing

The next step of our methodology is the analysis of system behavior and fault injection (test) result qualification. Embedded system dependability evaluation requires efficient testing strategies (e.g., SWIFI) to determine the performance of the system for various input and fault conditions. In order to assure proper test result qualification special methods must be defined to assess system behavior (outputs for various inputs) and classify it as correct, incorrect, time-outs, system exceptions, etc. Note, that in real-time systems, result qualification criteria can be complex because systems can have different dynamics due to used control algorithms.

The result qualification methods should correctly classify various possible responses of embedded systems. In real-time systems this issue is further complicated because a system, due to feedback properties, can have many correct trajectories for which some deviation is acceptable. The fault injection results for real-time systems can be qualified according to:

- full trajectory analysis of system's behavior,
- partial trajectory analysis,
- final objective – checking satisfaction of required conditions (e.g. reaching a destination by a controlled object with acceptable energy usage).

Here, we discuss only partial and full (trajectory) result qualification methods, and skip objective result qualification because it is application specific. The full trajectory method is one that analyzes all available data samples for the entire time interval of the simulated system behavior. The partial method on the other hand, analyzes only some subset of data samples in the result qualification process. Additionally, for each trajectory, we define boundaries of correct behavior (linear or nonlinear). If the trajectory is enclosed within the predefined boundaries the system behavior is considered as correct. For non linear trajectories a superposition of linear boundaries may not assure sufficient accuracy, so more complex non linear boundaries can be defined (or a composition of linear and non-linear).

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

In many dynamic systems, it can be difficult to define boundaries of correct behavior. In such cases, integrated square error (ISE) methods can be used. ISE method computes the least squared error between a reference and a measured value [9]. In this method, one wants to minimize the accumulated (i.e., integrated) error so that the measured system trajectory tracks the reference (i.e., expected) trajectory. Mathematically this can be expressed as *min*(ISE), where

$$\text{ISE} = \text{ ISE} = \int \left| \bar{f}(x) - f(x) \right|^2 dx$$

and $\bar{f}(x)$ is the reference (i.e., expected) function value and $f(x)$ is the measured function value. The main benefit of this method is that it analyzes the entire trajectory and computes the aggregated error.

Partial boundaries can also be used for result qualification. In those cases, we only analyze a subset of data samples from the trajectory to qualify system behavior as correct or incorrect. In our case study, we used a variation of this method (called the min-max window method) that qualifies system performance based on values of selected variables near the end of simulation time. However, this may not always be possible. In some real-time systems, selection of such variables can be difficult (i.e., all system variables are dynamic within the simulation time). In those situations, one may need to sample at various time intervals for example, at the beginning, in the middle, and at the end of simulation time. For each time interval, the result qualification process (i.e., the analyzer in Figure 1) can compute an average value of all data samples, and check if they are within a specified threshold margins. If this is the case for all selected time intervals, system behavior can be qualified as correct. However, the selection of thresholds can be difficult in practice, and may require a good knowledge of system dynamics.

The result qualification process is quite challenging in real-time systems that use the feedback property for object control. An important issue is to assure low time overhead of the qualification process. In practice, this can be difficult issue because in a highly dynamical system, analysis of all data samples can indeed require prohibitively large amount of time. Therefore, in less critical applications, it makes sense to compromise qualification quality for faster performance evaluation. This is especially true in SWIFI experiments, where millions of faults are injected, and for each fault, system performance must be evaluated.

To qualify the performance of our single-wheel (local) anti-lock braking case study, we used two most important parameters – the vehicle braking distance, and car final velocity. In our research, we also considered other parameters such as the wheel slip or applied brake torque, but eventually decided not to use them because they were more dynamic and thus computationally demanding to monitor. We decided to use less dynamic, monotonically increasing (braking distance) or decreasing (car's speed) parameters that described the system performance (i.e., car's stopping distance during braking and the final velocity at that distance). We discuss specific values of these parameters and the implementation of the result qualification in [23]. We also note that the result qualification for a distributed ABS system is substantially more complicated than for a non-distributed, single-wheel case. This is because the dynamics of the distributed system such as delays due to network communication, can introduce some randomness into control task computations that result

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

in a set of different but correct system trajectories (in fault-free conditions). Hence we have to take into account such randomness to perform correct result qualification.

3.3. Application instrumentation

An important issue in simulations is the application instrumentation so that dependability (SWIFI) experiments could be performed on real-time (distributed) systems. This instrumentation involved analysis and modification of a complex (over 10,000 lines of code) real-time simulator (TrueTime), adaptation of the controller model to this simulator, and separation of code for different controllers. All of these tasks required a significant amount of work and extensive verification before successful simulations could be executed.

Specifically, we translated our initial, functional and non-distributed system design to a task based, distributed design that we discuss in [25]. The design translation required:

- defining some timing parameters (task worst case execution times, activation periods, initial activation time offset), and selection of the network message priority (in case of event-triggered networks) or the communication schedule (in case of time-triggered networks),
- selecting appropriate simulation time granularity to match the environment and network communication dynamics,
- adding a task execution schedule monitor that collects data samples for result qualification,
- modifying the result qualification module (i.e., the analyzer in Figure 1) by setting slightly different min-max window size to account for different trajectory dynamics due to network message communication delays,
- adding functionality to simulate abstract faults discussed in [23] and [26].

Additionally, we modified the real-time simulator so that it could be used for SWIFI experiments. In this case, the modification consisted of an analysis of about 100 functions related to the real-time operating system kernel and network, and adding to them, simulation time and pointer references to simulation objects (e.g., controllers). Moreover, we substituted functions called by the Matlab process (i.e., *mdlInitializeSizes()*, *mdlInitializeSampleTimes()*, *mdlStart()*, *mdlInitializeConditions()*, *mdlOutputs()*, *mdlZeroCrossings()*, *mdlTerminate()*, etc.) with developed equivalents. This replacement was necessary because our SWIFI tool could not interface directly to the Matlab process. By replacing above function calls with our own functions, we successfully adapted the TrueTime simulator to the MSVC 2005 development environment, specified testing regions in the associated project source files, and eventually performed fault injection tests on the compiled source code.

Another practical issue of the application instrumentation was to use a transient fault injector (FITS) to activate different types of faults, namely abstract faults (see [23, 25, 26]). These abstract faults combine functional and dynamic fault models, and affect performance parameters important in real-time systems (i.e., message delay, task execution time, etc). As such, in our approach, we activated abstract faults via FITS, and the memory symbol map file (MAP) generated by the Microsoft Visual Studio development environment (on the x86 architecture). The symbol map file contains address offsets of all data objects used in the program. By knowing their variable addresses, we can select some of them and input them into FITS (by using injector's *Memory at...* fault injection location) to activate our abstract

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

fault functions. The FITS injector in this case practically provides fault triggering and result collection functionalities.

The time period within which the fault can be triggered is marked with two machine instruction sequences: *<int3, nop, nop>; <int3, nop, no>*. The first one marks the beginning of a test region, while the second one marks the end of the region. Normally, in transient fault injection these two marks are separated by a sequence of program instructions (the test region). However, in the abstract fault triggering, FITS uses the empty test region, to break the program execution, and modifies the selected memory location obtained from the MAP file. This modification then activates the source code of the abstract fault. Details of this activation method are explained in [23].

In addition to fault triggering, we also use FITS to collect statistics related to abstract and transient fault injection tests. When a program finishes its execution, it generates an output result that can be qualified as correct, incorrect, system timeout, and system exception. Each test output is then encoded by the programmer into an exit code, and FITS collects these exit codes to compute a statistical result for each fault injection experiment that usually consist of many tests. Such collection of results is very useful because it allows automatic result qualification in experiments that can require very large number of tests.

One practical conclusion of the application instrumentation is that it would be very useful if the transient fault injector FITS could be compiled as a dynamic link library that could be directly used in the Matlab/Simulink environment. This would eliminate the need to translate the functional design into the MSVC integrated development environment, and fault injection could be done directly in Matlab. Such direct implementation of FITS in Matlab would allow direct evaluation of functional designs that cannot be easily translated into C or C++ programming language code.

3.4. Fault space exploration

Transient faults can be generated pseudorandomely in space (memory cells, registers, etc.) and time [10, 24]. In case of abstract fault space exploration, we define and implement abstract faults (see [23, 25, 26]) that affect the functional and dynamic behavior of various system components (e.g., tasks, network MAC finite state machines, message transmission time, clock resolution, etc.) and use them in practical SWIFI experiments. Abstract faults provide the capability to investigate corner-case (i.e., rare) faults, that may be difficult to generate via random transient fault injection; the work presented in [19] confirms this idea. Moreover, since in model-based fault injection, the fault space is smaller than in target architecture program, methods such as abstract fault injection are useful to extend the explored fault space. Finally, abstract faults can also cover unknown fault models (e.g., due to a human design error).

In [23], we show that it is possible to correlate some abstract (task execution) faults with transient (latched) faults because both lead to excessive program execution times (i.e., program timeouts). However, in transient fault injection the program execution time is not specifically controlled, and thus it can be difficult to assess exactly how increasing program run-time affects other tasks running on the same microcontroller. On the other hand, in our abstract task execution time (TEF) fault model, we have good controllability over the execution time of the task, and thus can more accurately assess the effects of execution time faults on the application performance. For example, in [23], we show that task execution

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

scheduling performance can be correlated with braking distance measurements, in the anti-lock braking automotive case study.

3.5. Test scenario development

The final step in our dependability evaluation methodology is related to the development of test scenarios. The objective of tests was to:

- study the natural fault susceptibility of the original application (i.e., without any fault hardening mechanisms),
- identify faulty application behavior,
- analyze fault propagation in the system,
- perform statistical analysis (e.g., register activity, machine instruction distribution, etc.),
- improve application's natural fault susceptibility by applying selected software hardening mechanisms in critical and non-critical environment scenarios.

The developed dependability simulation methodology has been verified successfully in quite complex and representative studies related to the automotive anti-lock braking system controller, as opposed to other studies in the literature usually limited to some fragmented problem. In our work, we deal with a complete controller and environment models, wide spectrum of faults, and practical implementation issues. Next, we discuss some model details of our case-study.

4. Anti-lock braking case study

In this section, we present Anti-lock Braking System (ABS) as an embedded system case study used in the dependability analysis. Specifically, this section focuses on the control and environment models used in experiments.

The ABS system is popular in all modern cars and uses the widely applied Proportional Integral Derivative (PID) controller, hence it can be considered as a good real-time benchmark application for dependability experiments, the more that its dynamic is quite high and involves many parameters delivered from various sensors. As opposed to other studies of ABS systems (e.g. [28]), we consider wider class of faults (i.e., transient and abstract faults) that affect the PID controller. Moreover we consider a distributed version of ABS that functions with a real-time simulation tool. This tool effectively simulates the real-time operating system kernel (i.e., its scheduler) and media access network protocols; to our best knowledge, no other works performed such a deep study.

The ABS dependability study consists of the non-distributed, single wheel and distributed four-wheel cases. The non-distributed case was developed from dynamical models discussed in [20], it deals only with the controller algorithm (described in [24]). It was used to gain the necessary experience for developing more realistic 4 wheel simulations. The distributed case is an extension of the single wheel case, and implements a distributed control algorithm whose processing nodes communicate via event-triggered (CAN) or time-triggered (TTCAN) network protocol.

The distributed system uses a real-time simulator TrueTime [29] that was modified to support transient fault injection. By using TrueTime, we were able to perform more representative study because this tool allowed us to include main components (operating

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

system kernel and network) of a real system into the simulation. The modification of the simulator was necessary because it was originally developed in the MATLAB/Simulink context (i.e., it used Simulink call back methods) and not as a stand-alone tool. In order to be able to use this simulation tool in our fault injection experiments, we separated TrueTime from MATLAB run-time engine, and made it compatible with FITS (fault injector).

4.1. Local model

To study the fault susceptibility of ABS controller we first developed its program and the model of the external environment covering the behavior of the car in relevance to the road conditions. The modeling of ABS is based on mathematical equations describing Newton's 2nd law of motion defined separately for *x*, *y*, and z Cartesian coordinate axis. The ABS controller model has been defined using Matlab scripts for Simulink and is based on the mathematical model given in [20]. The developed Simulink model has later been transformed into C++ language for software fault injection based on SWIFI technology. In our case, we are mainly interested in the vehicle motion in the x direction.

Whenever a vehicle breaks, the resultant instantaneous net force acting on the car is less than zero. The objective of the anti-lock braking systems is to minimize the braking distance under the constraint of tire slip, and allow the driver to maintain safe vehicle control during braking. Tire slip occurs in situations where excessive braking force pressure is applied to braking pads while the friction force provided at surface contact point of the tire and the road is insufficient. For example, ice, snow or even water (often mixed with dirt and oil) covering the road surface can lead to insufficient friction force during heavy braking.

Exceeding the optimal braking force results in a dangerous wheel lock condition in which the angular velocity of the wheel is zero (i.e., the wheel stops to rotate), and the only friction force acting on the tire is the slip friction. The slip friction is usually much lower than non-slip friction and may result in long braking distance. Therefore, anti-lock braking system has been developed by Bosch in the 1970s, so that vehicle "slip" may be prevented in situations requiring sudden braking maneuver. Analyzing ABS dependability, we study the motion of only one wheel relative to the quarter vehicle body mass and road surface. This standard model known as the Quarter Vehicle Mass (QVM) model is also used in most other works that analyze the performance of the ABS.

The developed ABS controller unit is composed of two blocks: control logic block module and signal processing block. The input signals to the controller are *brake* signal (from the brake pedal), wheel angular velocity *omega*, and some constants specifying various mechanical parameters. These signals are obtained from the controller environment. There are only two controller output signals namely the *inlet* and *outlet* valve control signals. These signals force appropriate brake torque within the hydraulic mechanism. The controller monitors sensors, calculates critical parameters, and delivers control signals to actuators. While computing its outputs, the control algorithm exchanges information with a specific car behavioral model. The car behavioral model simulates the dynamics of the vehicle. Next, we discuss the distributed model which realistically represents ABS behavior.

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

4.2. Distributed model

This section discusses the extension of a single-wheel ABS model, namely the four-wheel ABS model. We decided to extend the single-wheel case in order to consider the kernel, and network protocols that are the foundation of modern, embedded distributed systems. Therefore, to perform such advanced system dependability evaluation, we decided to adapt the TrueTime tool supporting distributed and real-time control system simulation (in a single process running on a workstation) to the MSVC 2005 development environment for flexible fault injection.

Figure 2 presents the model of our distributed, Anti-lock Braking System. This model has been developed in C++ and consists of over 20,000 lines of source code; the previous, single-wheel, non-distributed version consisted of only about 2,000 lines of C++ source code. This increase in code size is due to the additional, separate code required for the remaining three ABS controllers, kernel, network, environment and the TrueTime simulator code. The TrueTime simulator requires about 12,000 lines of code, where the remaining 8,000 lines of code is the distributed ABS application.

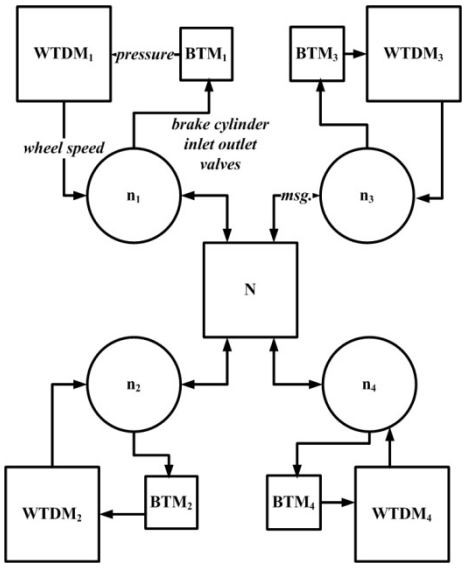

Fig. 2. The model of distributed ABS system tested with SWIFI

Rys. 2. Model rozproszonego systemu ABS testowanego za pomocą SWIFI

While developing the four-wheel ABS, we needed to assure separate code regions for all four nodes so that when injecting faults, we only disturb one particular controller. Hence the code of all simulation objects has to be disjoint so that when disturbing one object, the remaining are unaffected. We attempted to avoid common code effects by disabling the optimization features of the compiler and using different (symbol) names for the same functions to avoid possible code sharing. However, it is still possible that portions of the code could be shared between ABS controllers because compilers can introduce optimizations that limit redundant code within the binary image.

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

Code-sharing constraint can be relaxed with distributed processes. Such processes can be used to solve the code-sharing problem but in our case, this was infeasible because TrueTime code could not be easily converted to support multiple processes and would require a large amount of development resources to assure reliable simulations. We did not have these resources and thus used the single process approach.

The distributed ABS model instantiates four main modules replicated for all four wheels. These modules are:

- Wheel and Tire Dynamics Module ($WTDM_i$): responsible for simulation of the wheel dynamics based on suspension and associated quarter vehicle mass
- Brake Torque Modulator Module (BTM_i): responsible for simulation of the brake fluid pressure modulator
- Control Node (n_i): responsible for the simulation of an ABS microcontroller that contains the ABS controller object and the real-time operating system kernel
- Network Node (N): responsible for the simulation of a network protocol and interface

All controller objects within the network N communicate via a protocol-specific (e.g., CAN) messages to assure distribution of control information. The break pedal position (e.g. related to dynamic pressing) is provided by a sensor, which broadcasts it via the network as a message and this is implicit (not shown) in Figure 2. The three remaining simulation objects (i.e., n_i, BTM_i, $WTDM_i$) have one input and output. The n_i input is the wheel speed (angular) delivered by the $WTDM_i$ object and measured by the wheel speed sensor. The n_i output is a two bit-wide control line for the two valves (*inlet* and *outlet*) controlling fluid pressure in brake cylinders; this control line is an input to the BTM_i. The Brake Torque Modulator then computes the corresponding pressure force exerted by the brake pads on the wheel (based on the position of valves), and provides this information to $WTDM_i$ object to simulate car's dynamics (e.g., wheel angular and horizontal speed).

In our SWIFI experiments we used the local and distributed models to assess how a system responds to various types of faults, and checked if this response could be improved by using software fault hardening mechanisms. Next, we discuss some experimental results that focused on such analysis.

5. Experimental results

The main purpose of experiments presented in this section was to verify our dependability evaluation methodology discussed in section 3. Specifically, the objective of these experiments was to analyze how transient and abstract faults affect the performance of a real-time application whose functional models were implemented in a high-level programming language. This performance was evaluated for two different system configurations. In both configurations two main parts, namely the network and the controller were tested for fault susceptibility. In the first configuration, we evaluated the real-time application's natural fault susceptibility (i.e., no fault hardening mechanisms were used) to transient (latched and non-latched) and abstract (see [23–26]) faults. In the second configuration, we checked if fault-hardening mechanisms could improve the system's natural fault susceptibility to both fault types (transient and abstract). As such, we implemented and evaluated the system's fault coverage of software-hardening mechanisms

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

based on exception handling, assertions, static code redundancy, exact voter and a watchdog (see [23]).

In our research, we evaluated the dependability of two types of ABS systems. In the first case, we evaluated the dependability of a non-distributed, single-wheel ABS that used a standard Quarter-Vehicle-Mass (QVM) model to simulate car's dynamics [20]. The QVM model uses mass and spring model that represent the car's suspension; this model is interfaced with tire and road models. In the second case, we evaluated the dependability of a distributed, four-wheel ABS system that consists of four ABS controllers (one per each wheel) exchanging specific tire slip messages. These messages inform each controller on the braking performance of all tires. The distributed ABS allowed us to evaluate the effects of message delay and task execution time faults, on the application performance. This evaluation needed some adaptation of the real-time simulator TrueTime (see [29]).

5.1. Experiment set-up

The concept of the Software Implemented Fault Injector (SWIFI) relies on the software emulation of a fault during the run-time of the application under test. In this research FITS fault injector was used [8, 10]. It is based on standard Win32 Debugging API to control the execution of the software application under tests.

The objective of the performed simulation experiments was to analyse propagation of the fault effects in the system and their impact on the car braking process. Simulating faults we have used two strategies: random and selective. In the first approach we generated faults at random with equal distribution in space (fault locations: CPU registers, memory cells, etc.) and activation time. Such experiments give some general view on overall fault susceptibility. However, due to the limited number of injected faults we can skip some critical situations. Hence, we have developed selective fault injections targeted at specific areas. In particular, we used this for checking fault susceptibility of various internal variables.

In each experiment we injected 1000 bit-flip faults into the program's data memory (MEM), processor registers (REG), program's static code memory (CODE), the executed (dynamic) instructions (INST), and the processor's floating pointing unit (FPU), or a set of ABS controller parameters. For each fault injection test we then collected data on car trajectory and controller responses and compared that with the reference ones via a special result qualification (RQ) module (contained within analyzer shown in Fig. 1).

The result qualification enabled us to collect braking performance statistics from our SWIFI experiments and classify them as correct (*C*), incorrect (*I*), system exception (*S*) and timeouts (*T*). The correct (C) means that a program executing the ABS controller produced an expected (acceptable) trajectory. The incorrect (*I*) test means that the program provided unexpected (unacceptable) trajectory. The system exception (*S*) test means that some unhandled system/hardware exception was generated (e.g., illegal memory access). The timeout (*T*) means that the program exceeded the maximum expected execution time.

To qualify simulation results for the four-wheel ABS, we used measurements of individual wheel trajectories that had to fit within result qualification window, with some tolerance value ε. In case of the qualification window, each individual wheel stopping distance had to be between $d_{\min} = 14.0$ and $d_{\max} = 16.5$ m, the car final velocity between $cwv_{\min} = 0$ and $cwv_{\max} = 1.5$ m/sec, and wheel slip between $ws_{\min} = -0.9$ and $ws_{\max} = -0.0005$ (no units) within the simulation time range $t_{\min} < t < t_{\max}$. In our

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

experiments, t_{min} and t_{max} window variables were set to 1.49 and 1.99 seconds respectively. We selected the trajectory tolerance value $\varepsilon = 10\%$, which means all four stopping distances and final velocity trajectories for all four wheels can not differ by more than 10% relative to each other. This tolerance value along with window size parameters were derived from our reference (golden-run) ABS simulation experiments, in which the car's initial velocity was 60 km/hr, the vehicle mass was 1200 kg, and axle and rim mass was 35 kg and tire parameters are given in [20].

We present some examples of computer simulation experiments performed with the non-distributed and distributed ABS models that illustrate what can be achieved with our methodology. Specifically, we discuss experiments showing the effect of exception handling. We also present how application-specific software hardening mechanism (e.g., assertions) can increase application's resistance to transient faults. Finally, we discuss experimental results for various program versions.

5.2. Exception handling

The objective of the experiment discussed in this section was to study how exception handling can affect system's ability to detect and handle transient faults. To achieve this, we used our simulation methodology discussed in section 3, and applied it to the ABS system. As such, we checked how the system responds to transient faults affecting its microcontroller registers (REG), program data memory (MEM), floating-point unit (FPU), static code (CODE) and dynamic instructions (INSTR). We then compare system behavior for program versions that did not implement exception handling with behavior of versions that used it. Additionally, we tested controller versions with built in software hardening mechanisms activated whenever an exception was raised (e.g., *divide-by-zero*, *illegal-memory-access*, *overflow*, *underflow*, etc.) by the operating system.

In Figure 3, we present experimental results for integrated exception handling. The experiment was performed on a local model of the ABS controller, where three versions were studied. The first (reference) version did contain neither exception handling nor any other fault hardening mechanisms. The second (hardened) version implemented exception-handling code together with *_fpreset()* function that reset the floating point unit whenever an exception was raised by the system. The third (hardened) version used exception handling together with code re-loading (via a dynamically linked library) that allowed the use of a spare controller code image when the primary static image was disturbed.

For each experiment involving tests of one version, we injected on average 1000 faults (randomly distributed in time and space) into each processor resource, observed system's behavior for each fault, and qualified this behavior as correct (*C*), incorrect (*I*), timeout (*T*), or system exception (*S*). Figure 3 shows results for the three versions of the tested application. Results of the first version (without fault hardening mechanisms) are shown in the first column (from left) for fault locations REG, MEM, FPU, CODE, INSTR. Results for the second version (with exception handling and *_fpreset()* function enabled) are shown in the second column, while the results for the version with additional code re-loading is shown in the third column. The results show that exception handling can significantly increase the number of correct program executions because exceptions are detected and handled correctly. However, for the two versions with exception handling enabled, we also recorded more (from 3–15%) incorrect program executions when compared with the non-hardened version; this is due the fact that even in hardened program versions some

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

faults were not recovered. As such, there is a need for additional mechanisms that can handle such faults correctly. In the next section, we illustrate one possible improvement in fault coverage for selected memory locations storing critical ABS controller parameters.

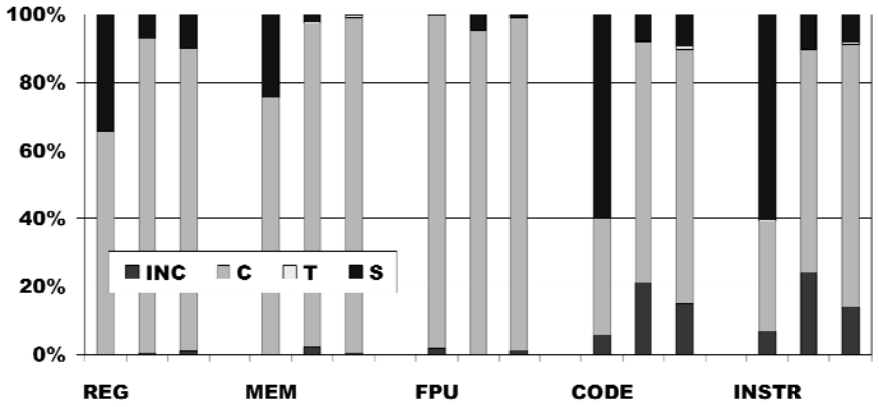

Fig. 3. Fault susceptibility of ABS with integrated exception handling

Rys. 3. Wrażliwość systemu ABS na błędy z obsługą wyjątków

5.3. Selective assertions

In this fault injection experiment, we selected a set of controller parameters and observed the system behavior in the presence of parameter faults. We analyzed the ABS system with no fault hardening mechanisms (Tab. 1 – version A) and an enhanced version with built-in simple software assertions (Tab. 1 – version B). These assertions were targeted at critical parameters and their effectiveness was checked experimentally.

In this approach an important issue is the selection of critical parameters. Main parameters selected for experiments were wheel slip threshold (*Slip Tresh.*), low-pass filter time constant (*Filter TConst.*), low-pass filter amplification gain (*Filter Gain*), integrator sample time (*Int. Sample Time*), and effective tire radius (*Tire Radius*). The significance and typical values of these ABS controller parameters are discussed in [20].

In Tab. 1, we show parameter oriented fault injection results for a controller with CAN network without and with fault hardening mechanisms (application specific assertions). In each experiment we computed the total number of correct (*C*), incorrect (*INC*), system exceptions (*S*) and program timeouts (*T*) to determine the performance of the system in the presence of faults. Version A shows non-hardened controller results (i.e., distribution of result categories in percents) for a specific parameter (denoted with acronym as given above). Version B shows assertion-hardened controller results that protect a specific parameter shown in Table 1. Errors detected by assertions recover appropriate parameter values.

For the five selected parameters we injected a total of 5000 bit-flip faults that disturbed the controller during the initial (90–900) control iterations (testing region). We disturbed ABS parameters during the initial stages of controller activity because in this time interval ([9.9 ms, 99 ms], one iteration is equivalent to 0.11 ms) most of the braking force must be

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

generated to stop the vehicle safely. The results of Table 1 show that using simple assertions (10 static machine instructions required for each parameter), one can significantly improve controller's ability to tolerate faults affecting its critical parameters. Moreover, only a small percentage of injected faults produce incorrect results and the remaining faults are either detected or tolerated. For some parameters (e.g., *Slip Tresh.*) the dependability enhancement is very high with low static and dynamic code size overheads (less than 5% when compared to the primary version).

Table 1

Improving fault robustness of ABS with application-specific assertions (A/B – percentage for non hardened and fault hardened version – bold figures)

Parameter	Test results (A/**B**: A – non hardened version, **B** – fault hardened version)			
	C	INC	S	T
Slip Tresh.	55%/**98%**	45%/**0%**	0%/**0%**	0%/**2%**
Filter TConst.	91%/**97%**	9%/**0%**	0%/**0%**	0%/**3%**
Filter Gain	78%/**98%**	16%/**2%**	0%/**0%**	6%/**0%**
Int.Sampl.Time	82%/**99%**	16%/**0%**	2%/**0%**	0%/**1%**
Tire Radius	96%/**99%**	4%/**0%**	0%/**1%**	0%/**0%**

5.4. Fault resistance and program design

In this section we discuss program's design influence on its susceptibility to transient faults. Specifically, the objective of this experiment was to compare how transient faults can affect a program that passes data-by-value or reference (i.e., via pointer values). This is quite interesting, because it can give guidance for software development that is more resistant to faults.

The results of our experiment measuring fault susceptibility of ABS to transient fault affecting controller's resources are presented in Figure 4. In the experiment, results for two program versions are shown. The pointer-based version is labeled with *PTR*, whereas the pass-by-value version does not have this label. Similarly to previous experiments, this experiment injected 1000, randomly distributed, latched memory faults into processor registers (*REG*), data memory (*MEM*), program static code (*CODE*), dynamic instructions code (*INST*) and the floating point unit (*FPU*). For each fault injection test, program output was qualified as correct (*C*), incorrect (*INC*), timeout (*T*) or exception (*S*).

Figure 4, shows aggregated distribution of results for all fault injection tests for the two program versions (pointer-based and pass-by-value). The results show that pointer-based design generates more exceptions and less incorrect results than the pass-by-value by 1–10% (depending upon fault location). This is somewhat expected, since whenever a transient fault affects data's memory location, it is highly probable that this data is a pointer variable that gets redirected to an invalid memory location, thus generating an exception (e.g., *illegal-memory-access*). In case of pass-by-value, a similar fault would most likely result in a direct corruption of data later used for computation of control signals. As such, the experiment shows that pass-by-value designs may be less susceptible to exceptions, but once affected by a fault, can generate more incorrect results than

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

pass-by-reference designs. As such, there is a need for more effective fault hardening mechanisms such as software assertions.

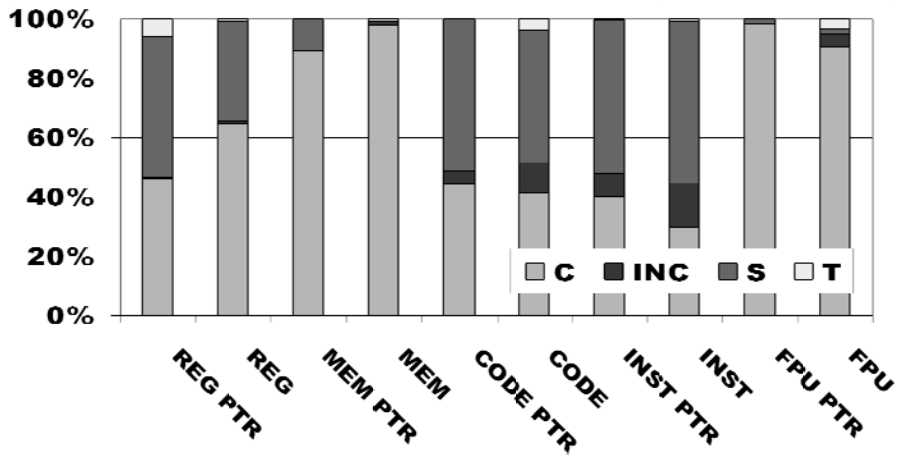

Fig. 4. Fault susceptibility of ABS for various program designs

Rys. 4. Wrażliwość systemu ABS na błędy dla różnych projektów programu

6. Conclusions

The developed methodology of dependability evaluation integrates software implemented fault injection (SWIFI) with various simulation techniques within Windows architecture platform. As compared with other SWIFI approaches, we have assured higher flexibility in combining real application programs with functional, interacting models of controlled objects and system environment. It is supported with an original test result qualification module and allows us to deal with larger fault space. To assure good test controllability various technical problems has been resolved in particular code isolation issues discussed in the paper. The methodology has been verified in the analysis of a complex and representative real-time control application (Anti-lock Braking System). Our experiments revealed that:

- system's fault susceptibility depends on its dynamics, control phase, and ability to mask faults via the feedback property,
- effective exception handling can cover over 90% of transient faults,
- more complicated software fault-hardening mechanisms require large static and dynamic overhead exceeding 100% of the protected code size,
- simple software mechanisms such as assertions have low overhead (less than 5% in our ABS case) and can provide up to 99% fault coverage for selected memory locations,
- efficient result qualification and code separation is essential to interpret simulation results,
- pointer-based programming may be more resistant to transient faults than pass-by-value provided that exceptions (related to faults) are handled effectively.

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

In the future, we plan to verify other control applications with different object characteristics (e.g., reaction time) and check fault susceptibility for various control algorithms (e.g., linear, non-linear). This research will be combined with the problem of mapping structural (low-level) faults into (high-level) abstract faults. This can facilitate extending the fault space of the experiments.

We express our appreciation to P. Gawkowski for his advice on using FITS for Anti-Lock Braking System fault injection experiments.

Literatura

[1] Aidemark J., et al., *GOOFI: Generic Object-Oriented Fault Injection Tool*, Proc. International Conference on Dependable Systems and Networks DSN 2001, Goteborg, Sweden 2001, 668-668.

[2] Arlat J., et al., *Comparison of Physical and Software-Implemented Fault Injection Techniques*, IEEE Trans. on Computers, Vol. 52, No. 9, 2003,1115-1133.

[3] Clark J.A., Pradhan D.K., *Fault Injection a Method for Validating Computer--System Dependability*, IEEE Computer, Vol. 28, No. 6, June 1995, 47-56.

[4] Cunha J.C., et. al., *A Study of Failure Models in Feedback Control Systems*, Proc. International Conference on Dependable Systems and Networks DSN 2001, Goteborg, Sweden 2001, 314-326.

[5] Cunha J.C., *Reset-Driven Fault Tolerance*, Proc. 4th European Dependability Computing Conference EDCC-4, Toulouse, France 2002, 102-120.

[6] Dawson S., Jahanian J., Mitton T., *ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementations*, Proc. IEEE International Symposium on Computer Performance and Dependability, Urbana-Champaign, USA 1996, 56.

[7] Drebes R.J., et al., *ComFIRM: a Communication Fault Injector for Protocol Testing and Validation*, Proc. 6th Latin American Test Workshop LATW, Salvador, Brazil 2005.

[8] Gawkowski P., et. al., *Dependability of Explicit DMC and GPC Algorithms*, Proc. International Multiconference on Computer Science and Information Technology, Wisła, Poland 2007, 903-912.

[9] Gawkowski P., et. al., *Software Implementation of Explicit DMC Algorithm with Improved Dependability*, Springer: Novel Algorithms and Techniques in Telecommunications Automation and Industrial Electronics, 2008, 214-219.

[10] Gawkowski P., Sosnowski J., *Using Software Implemented Fault Inserter in Dependability Analysis*, Proc. 9th Pacific Rim International Symposium on Dependable Computing PRDC 2002, Tsukuba-City, Japan 2002, 81-88.

[11] Goswami K.K., Iyer R.K., Young L., *DEPEND: A Simulation-Based Environment for System Level Dependability Analysis*, IEEE Transactions on Computers, Vol. 46, No. 1, 1997, 60-74.

[12] Guthoff J., Sieh V., *Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method*, Proc. 25th International Symposium on Fault Tolerant Computing FTCS-25, Pasadena, CA, USA 1995, 196-206.

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

[13] Herout P., Racek S., Hlavicka J., *Model-Based Dependability Evaluation Method for TTP/C Based Systems*, Proc. 4th European Dependability Computing Conference, Toulouse, France 2002, 271-282.
[14] Jenn E., et al., *Fault Injection into VHDL Models: The MEFISTO Tool*, Proc. 24th International Symposium on Fault Tolerant Computing FTCS-24, TX, USA 1994, 66-75.
[15] Kanawati G.A., Kanawati N.A., Abraham J. A., *FERRARI: A Flexible Software-Based Fault and Error Injection System*, IEEE Transactions on Computers, Vol. 44, No. 2, 1995, 248-260.
[16] Kopetz H., Grunsteidl G., *TTP – A Time-Triggered Protocol for Fault--Tolerant Real-Time Systems*, Proc. IEEE 23rd International Symposium on Fault Tolerant Computing FTCS-23, Toulouse, France 1993, 524-533.
[17] Kopetz H., *Real-Time Systems – Design Principles for Distributed Embedded Applications*, Kluwer Academic, Netherlands 1998.
[18] Muranho J., et. al., *Failure Boundness in Discrete Applications*, Proc. 3rd Latin--American Symposium on Dependable Computing, Morella, Mexico 2007, 160-169.
[19] Pattabiraman K., et al., *SymPLFIED: Symbolic Program Level Fault Injection and Error Detection Framework*, Proc. International Conference on Dependable Systems and Networks DSN 2008, Anchorage, Alaska, USA 2008, 472-481.
[20] Rangelov K., *Simulink Model of a Quarter-Vehicle with an Anti-Lock Braking System*, Research Report, Eindhoven University of Technology, 2004.
[21] Rodriguez M., et al., *MAFALDA: Microkernel Assessment by Fault Injection and Design Aid*, Proc. International European Dependability Computing Conference EDCC-3, Prague, Czech Republic 1999, 143-160.
[22] Stott D., et al., *NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors*, Proc. IEEE International Symposium on Computer Performance and Dependability, Chicago, USA 2000, 91-100.
[23] Trawczynski D., *Dependability Evaluation and Enhancement in Real-Time Embedded Systems*, Ph.D. Thesis, Warsaw University of Technology, Warsaw, Poland 2009.
[24] Trawczynski D., Sosnowski J., Gawkowski P., *Analyzing Fault Susceptibility of ABS Microcontroller*, Proc. International Conference on Computer Safety, Reliability, and Security SAFECOMP'08, Newcastle, UK 2008, 320-372.
[25] Trawczynski D., Sosnowski J., Gawkowski P., *Testing Distributed ABS System with Fault Injection*, Proc. International Joint On-Line Conference on Computer, Information, and System Sciences, and Engineering – CISSE'09, On-line Conf., 2009.
[26] Trawczynski D., Sosnowski J., Zalewski J., *A Tool for Databus Safety Analysis Using Fault Injection*, Proc. International Conference on Computer Safety, Reliability, and Security, SAFECOMP'06, Gdansk, Poland 2006, 261-275.
[27] Vinter J., et al., *Experimental Dependability Evaluation of a Fail-Bounded Jet Engine Control System for Unmanned Aerial Vehicles*, Proc. International Conference on Dependable Systems and Networks DSN 2005, Yokohama, Japan 2005, 666-671.
[28] Skarin D., *Software Implemented Detection and Recovery of Soft Errors in a Brake-by-Wire System*, Proc. 7th European Dependability Computing Conference – EDCC 2008, Kaunas, Lithuania 2008, 145-154.

BPK BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

[29] Henriksson D., Cervin A., Arzen K., *TrueTime: Real-Time Control System Simulation with MATLAB/Simulink*, Proc. Nordic MATLAB Conference, Copenhagen, Denmark, October 2003.
[30] Zamli K.Z., et al., *An Automated Software Fault Injection Tool for Robustness Assessment of Java COTs*, Proc. of EEE ICOCI Conference, 2006.
[31] Li M.-L., et al., *Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design*, Proc. of ASPLOS'08, ACM 2008, 265-276.

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ