Analyzing Fault Susceptibility of ABS Microcontroller

Dawid Trawczynski, Janusz Sosnowski, and Piotr Gawkowski

Institute of Computer Science, Warsaw University of Technology,
Nowowiejska 15/19, 00-665 Warsaw, Poland
{d.trawczynski,jss,gawkowsk}@ii.pw.edu.pl

Abstract. In real-time safety-critical systems, it is important to predict the impact of faults on their operation. For this purpose we have developed a test bed based on software implemented fault injection (SWIFI). Faults are simulated by disturbing the states of registers and memory cells. Analyzing reactive and embedded systems with SWIFI tools is a new challenge related to the simulation of an external environment for the system, designing test scenarios and result qualification. The paper presents our original approach to these problems verified for an ABS microcontroller. We show fault susceptibility of the ABS microcontroller and outline software techniques to increase fault robustness.

Keywords: Fault injection, fault tolerance, safety evaluation, real-time embedded systems, automotive systems.

1 Introduction

Recently, in automotive industry, electronic embedded systems are gaining much interest resulting in steady increase of devices controlling various car functions involved in airbags, active brakes, engine control and x-by-wire operations ([1-6] and references). These applications result in quite complex microcontrollers for which fault occurrence cannot be neglected. In particular, soft errors are becoming real problem. Hence, dependable operation of electronic control devices is a crucial point and appropriate safety norms have to be assured e.g. IEC 61508 [7] or AUTOSAR [8]. This can be achieved with various redundancy techniques as well as specific error recovery procedures ([3, 9-10] and references). An important issue is to analyze the effectiveness of the proposed solutions for various classes of faults. In particular we have to deal with permanent, intermittent and transient faults. In practice, transient faults (due to electromagnetic interference, power brownouts, and environmental disturbances) are dominating, so we are mostly interested in this class of faults.

Many approaches to analyzing fault susceptibility were proposed ([11-15] and references therein). They base on formal methods (functional analysis, fault tree and failure mode effect analysis) and various simulation experiments covering specific fault models at different abstraction levels. Most simulation techniques related to automotive systems rely on a simulation model for the entire considered system (usually in Matlab/Simulink and TrueTime [14]) enriched by some fault injection capabilities [1,15]. Typically, faults are injected at some abstract level e.g. selected state variables, abstracting from the implementation, or they are targeted at some specific

M.D. Harrison and M.-A. Sujan (Eds.): SAFECOMP 2008, LNCS 5219, pp. 360–372, 2008.

problems. In [4] the authors analyzed the impact of CAN network bandwidth (effects of the delays and jitter resulting from the use of a shared bus) on car suspension control performance, packet suppression faults, and sensor faults. The simulation experiments in [1] were targeted at high level mathematical model of a car suspension control developed in Matlab/Simulink with embedded fault injection functions.

In our approach we are closer to the real implementation and faults, which are well emulated by the software implemented fault injection (SWIFI) tool at the level of binary code of the evaluated system. SWIFI simulates faults in the real system by disturbing (e.g. performing bit-flips) the states of processor registers and memory cells (used for storing the program code, data and stack). The fault injection moments and locations can be specified explicitly or in a pseudorandom way. The fault effects are analyzed by comparing the behavior of the disturbed program with the reference execution (golden run) with no faults. The SWIFI approach is a popular and widely used dependability evaluation method for classical computational programs [11,13,16,17]. Adapting this technique to reactive control subsystem is a kind of challenge, due to the problem of taking into account the interactions of various electronic and mechanical subsystems as well as the impact of the environment e.g. driver reactions, road and weather conditions.

To qualify the controlled object's response to faults we have to define some performance parameters that describe the quality of the performed task by the controller in request of the driver or other car sub-circuits. The measured deviation of the analyzed parameter from the nominal values gives an indication on performance loss or even critical and unacceptable situations. A large class of embedded systems used in automotive or other industrial applications relates to feedback control of physical systems. Such systems usually operate in a cyclic way. Typically they get signals from sensors, process them and deliver output signals to the actuators. The control algorithm may take into account system deviation from the correct behavior (due to external disturbances or even microcontroller faults) and compensate detected error by adjusting newly calculated outputs. In this process an important issue is to meet time requirements while producing output signals.

To meet dependability expectations various techniques can be used that are based on fail-silence property. Duplex systems built by pairing two subsystems with continuously compared output signals, triplicated systems with voting or cheaper designs with limited or no hardware redundancy all can, to a certain degree, exhibit the fail-silence property ([17] and references). In any case, the analysis of fault effects and error propagation for simplex systems is important since often duplex implementation is not cost effective for an application. It is worth noting that for many applications a single temporary malfunction of the controller is not critical due to the natural inertia of the controlled system. Moreover, a simple error recovery performed after fault detection using the available idle time of the microcontroller (so as to not exceed real time requirements) maybe also effective. These features have to be validated.

In the paper we present an original methodology of analyzing fault susceptibility using SWIFI fault injector FITS [9] with appropriate adaptation to reactive systems. This methodology has been verified for the anti-lock microcontroller, and the gained experience can be easily extended for other reactive systems. We have also performed similar experiments with robot and alcohol rectification microcontrollers [16]. Section 2 describes the ABS microcontroller operation in relevance to the behavior of

the braking system and the car. Section 3 outlines the fault injection platform and its adaptation to the required test scenarios. Experimental results are discussed in section 4. They show the impact of the faults on the performance of the ABS controller. The last section presents the conclusion and suggestions for the future research.

2 ABS Model

To study the fault susceptibility of the ABS controller we have 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 [18] 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 [18,19]. The developed Simulink model has later been transformed into C++ language for software fault injection based on SWIFI technology. We are mainly interested in the vehicle motion in the *x* direction. Whenever a vehicle brakes or accelerates the resultant instantaneous net force is lower or greater than zero. The objective of the anti-lock braking systems is to minimize the braking distance under the constraint of the tire slip. The tire slip occurs in situations where excessive braking force pressure is applied to braking pads while the friction force provided at the surface contact point of the tire and the road is insufficient. Exceeding the optimal braking force results in tire slippage, and in extreme may lead to tire locking. For the tire lock, the angular velocity of the wheel is zero, and the only friction force acting on the tire is the slippage friction. The slip friction is usually much lower than non-slip friction and may result in excessively long braking distance. Therefore, the anti-lock braking system has been developed by Bosch [18] in the 1970s, so that vehicle "slip" may be prevented in situations requiring sudden braking maneuvers. Analyzing ABS dependability we study the motion of only one wheel relative to the quarter vehicle body mass and the road surface.

The developed ABS controller unit (MCU) is composed of two blocks: the control logic block module (CLB) and the signal processing block (SPB). 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 shown in Fig. 1. The controller (MCU) monitors sensors, calculates critical parameters and delivers control signals to actuators according to the algorithm outlined in section 2.3. 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. The developed software is time triggered, a fixed sequence of tasks is activated periodically. There is no hardware replication. The behavior of the car is modeled as the controller environment. In the sequel we present a more detailed description.

2.1 ABS Controller

The control logic block generates two output binary signals: the *inlet* and *outlet* valve control signals. They are connected directly to the brake torque modulator module (BMM). This module is directly responsible for modulating the brake fluid pressure in individual brake lines. The binary TRUE (1) signal at the *inlet* output port commands the brake line valve to remain closed, whereas the binary *outlet* valve, depending on the situation, can be either closed - FALSE (0) or open – TRUE (1) at this time. Closing the *outlet* valve maintains current brake line pressure resulting in a constant torque, whereas opening the valve decreases the line pressure thus reducing the brake torque. Similarly, the binary FALSE (0) signal at the *inlet* valve commands this valve to be open thus increasing brake line pressure under the constraint that the outlet valve is in the closed position. Properly functioning control block module, under the condition of excessive tire slip, will generate a modulated sequence of pulses that increase, decrease or maintain brake line pressure. The concept of the used control algorithm is described in section 2.3.

The control logic block (CLB) accepts four binary control signals and its inputs are coupled indirectly through two OR gates to the signal processing block (SPB). The four input ports of the block are *decrease*, *hold*, *increase*, *stop decrease* and they either cause opening or closing of the associated *inlet* and *outlet* valves. Although the control action seems straight forward, the "tricky" part of the ABS control algorithm lies in the amount of time an inlet or outlet valve is either in the open or close position. This time effectively generates the necessary brake torque. The torque is a result of the brake line fluid pressure which is transmitted to the brake pads. The pads act on the wheel rotor (brake disc) surface to generate necessary friction torque. Therefore, a constant input to the logic control block produces a sequence of time varying output values forming a duty cycle varied pulse. This duty cycle modulated pulse is directly responsible for the pumping action of brakes in ABS. Its operation is described in [18]. The CLB block co-operates with the signal processing block(SPB).

The SPB block generates various signals needed to identify the state of the wheel. This block outputs two groups of binary signals. The first group is related to monitoring wheel acceleration and signaling crossing three thresholds: $-a$, $+a$ and A. The second group comprises 3 pairs of binary signals specifying the direction of crossing the above mentioned thresholds: in increasing (*pos.slope*) or decreasing (*neg slope*) direction. Moreover, SPB delivers the wheel slip coefficient *slip*, and the ratio of wheel angular acceleration to velocity *Om_dot/Om.* All these signals enable the logic controller (CLB) to determine if the tire has entered or is near the non-optimal friction region. If it happens, the wheel brake line is modulated as to bring the tire back to its optimal "friction" state. The computation of these signals and associated $A/+a/-a$ thresholds are given in [18]. The input signals to SPB block are the *vhvel*, *r_eff*, *omega*, and *brake* signals. The *vhvel* is the vehicle horizontal velocity (calculated by car dynamics module, described in the sequel), and *r_eff* is the effective rolling radius of the tire. This radius is a function of the tire stiffness and the normal force acting at the tire contact point. The *omega* is the angular velocity of the wheel and *brake* is a binary signal that is TRUE (1) whenever a driver presses the brake pedal.

2.2 ABS Environment Model

The ABS environment relates to three modules: brake modulator, tire and wheel dynamics module, and car dynamics. The brake modulator module (BTM) generates a real value of the simulated brake torque applied at the wheel rotor (disc brake). This module models a physical device and is represented in Fig. 1 as hydraulic pressure modulator. As the inlet valve remains open, the brake torque modulator integrates a constant rate of torque increase. When the integrator output exceeds the maximum brake torque that a brake pad may generate, the output of the integrator is saturated and kept constant. If the brake pedal is depressed by the driver, the torque modulator generates zero torque. This is accomplished by resetting the output of the integrator. In summary, BTM generates an appropriate brake torque as a function of inlet, outlet valve signals and the brake pedal state.

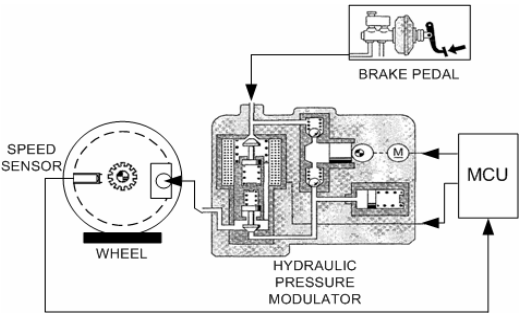

Fig. 1. Block diagram of the ABS brake system with the wheel speed sensor, hydraulic pressure modulator and electronic control unit (*MCU*)

The tire and wheel dynamics module (TWDM) is responsible for simulation of the wheel angular velocity *omega*. This value is generated based on two inputs – the *slip* (delivered by CLB) and applied *brake torque* (delivered by BTM). Additionally, the wheel angular velocity is computed based on an initial wheel velocity, polar moment of inertia of the wheel and tire, unloaded tire radius, vertical tire stiffness, effective tire radius, and normal force due to vehicle mass. These parameters are defined in [18,19]. Generally, as the slip value and brake torque increase, the wheel lock condition can be reached (the angular velocity of the wheel is zero). The controller therefore must adjust the brake torque to avoid the "wheel lock" state.

The car dynamics module (CDM) calculates the vehicle horizontal acceleration (*hac*) and velocity (*hvel*), and the *vehicle stopping distance* based on only two inputs: the wheel angular velocity *omega* and *brake status* signal. CDM calculates these signals and simulates the motion of the vehicle in the x direction taking into account the following parameters: vehicle mass, axle and rim mass, initial body translational velocity, initial axle translational velocity, tire belt translational stiffness, tire belt translational damping, vehicle translation dumping suspension, vehicle translation stiffness suspension, stop velocity, damping of translation, and normal force at the tire contact point. These parameters are defined in [18,19].

2.3 ABS Control Algorithm

The concept of the control algorithm is based on mathematical models from [18]. This concept has been implemented in software for x86 platform. The correctness of the implemented algorithm has been verified in many simulation experiments involving the developed ABS controller and the environment model. The objective of the algorithm is to decide what corrective action needs to be taken during excessive braking maneuver. The key variable of the algorithm is the wheel slip. This variable tells the algorithm if the wheel-tire system is in the normal (optimum friction) or abnormal (less than optimal friction) operating region. If the threshold slip is exceeded the tire is in the abnormal region and some corrective action needs to be taken.

The algorithm operates in an iterative way with specified time slot (0.1 ms) for each iteration. This time slot assures sufficient accuracy while controlling the brake mechanism. If an excessive wheel slip is detected during braking, the algorithm closes *inlet* and *outlet* valves of the brake torque modulator module (BTM) to maintain a constant brake pressure. The algorithm then measures the slip criterion during the subsequent iteration (time slot) and if the slip is still exceeding the required threshold, the algorithm commands the *outlet* BTM valve to open, thus reducing the torque rate at a constant rate measured in units of Nm/s. In the next phase, as the wheel speeds-up, the algorithm measures the wheel peripheral acceleration $+a$ and compares this measurement with a predetermined acceleration threshold. If the threshold is not exceeded the algorithm keeps the outlet valve open. In another case the outlet valve is closed and brake pressure is maintained constant.

In the last phase, the algorithm checks again the peripheral acceleration of the wheel to see if it exceeds the A threshold. This threshold determines when the braking torque should be increased again to maintain the safe braking action. As soon as this threshold is exceeded or a negative slope is detected in the peripheral acceleration of the wheel (here the A threshold can not be exceeded), the brake torque is increased by closing the outlet valve and opening the inlet valve of the BTM. The pressure is increased at a constant rate, either 2533 Nm/s or 19000 Nm/s, if ABS was activated for the first time in the braking interval. The first phase of ABS braking relates to the interval immediately after the driver has engaged the brake pedal. During this interval, the maximum brake pressure is applied until the slip value does not exceed the allowed maximum threshold. After the threshold has been exceeded, further brake torque increase occurs at a reduced or modulated rate. In this second control phase, the algorithm ensures that the possible wheel lock condition is avoided. The general structure of the algorithm is given below for typical values of some parameter thresholds (they can be adapted to other test scenarios):

```
WHILE (brake == TRUE AND vehicle_velocity > 1.5){
     inlet = OPEN
     outlet = CLOSE
     IF (first_braking_phase == TRUE) {
           brake_torque_increase_rate = 19000 Nm/s
     } ELSE {
     /*set second braking phase torque increase rate*/
     brake_torque_increase_rate = 2533 Nm/s}
     IF (current_vehicle_slip > 0.2) {
           /*close the BTM inlet valve*/
```

```
            inlet = CLOSE
            DO {brake_torque_decrease_rate = 19000 Nm/s}
            WHILE (wheel_acceleration < 0)
            outlet = CLOSE
            DO {outlet = CLOSE}
            WHILE          (wheel_acceleration<3          AND
      neg_slope_wheel_acceleration == FALSE)
      }
}
```

3 Fault Simulation Platform

To analyze fault effects in the ABS controller we use software implemented fault injector FITS [9], which has been adapted to deal with real-time and reactive systems. The fault injector treats the ABS controller and car environment as an integrated application. Faults are injected by disturbing the states of processor registers or RAM locations (storing program code and data).

Each fault injection called *test* needs the execution of the application and simulating a fault at an appropriate fault triggering moment. The fault triggering moment is correlated with the program instruction i.e. its location address and execution iteration (appearance number). The fault injection (fault triggering moment and its location) can be either specified directly by the user or generated in an automatic way e.g. according to pseudorandom strategy. In the pseudorandom strategy we specify only the number of injected faults and some indications on fault location, fault type (bit flip, bit setting, resetting and bridging), fault duration, etc. This process has to be done for each test scenario (i.e. input data). The fault triggering moments may be restricted to specified program modules or even code address ranges. Fault location can be defined explicitly (e.g. specific register such as EAX or EBX, RAM memory address) or pseudorandomly within a selected group of registers, memory code or data area. Similarly, the fault type can be defined explicitly (e.g. bit flip in a specified position) or pseudorandomly within specified bit areas and related to a fixed number of faults (e.g. m-bit flips). Depending upon the goal of the analysis we can either generate the most stressing fault injection scenarios (to find critical points e.g. in specified code areas) or assure the pseudorandom selection of fault triggering moments with equal distribution within the tested code space (static strategy) or within the time of the application execution (dynamic strategy).

In the performed experiments we specify fault-triggering moments and fault locations related only to the analyzed ABS controller. The system environment is not disturbed. For each injected fault (a test) we check the system behavior. Test results are identified in relevance to the reference execution of the analyzed application - so called the golden run. The golden run delivers GR log with the registered information on the dynamic image of the application execution. In addition, it comprises some statistical data related to the number of writes, reads, state changes for each CPU register, register activity [9], etc. (they are not encountered in other SWIFI injectors). This is helpful while profiling the experiment or analyzing experimental results.

In general, test results can be qualified as: C – correct result, INC – not correct result, T – time-out, S - system exception (e.g. access violation, invalid opcode, memory misalignment or parity errors, overflow), U – user message (generated by the application). We have to define some procedure qualifying C and INC results. It can be done in a general way or targeted at the considered application. In calculation oriented applications (mostly considered in the literature [13]) the result analysis is simple and coarse-grained e.g. binary qualification based on the comparison with the final correct result. In real time applications the result qualification usually is more complex due to the fact that we have to analyze the output signals trajectories in time. Moreover, different tolerance margins and incorrect behavior severity levels can be attributed. This may lead to fined-grained result qualification with more detailed information e.g. the file comprising the generated output results of the application. We resolved this problem by defining a special result qualification module coupled to the fault injector and the model of the controlled object (environment).

In the case of the ABS controller the test result analysis can be performed by selecting some output control variables (e.g. control signals of a brake) and comparing their trajectories with the non-faulty run. The comparison can be based on calculating the mean square error. Another approach is to analyze the brake effect using two main safety parameters – the vehicle stopping distance and its final translational velocity. The vehicle stopping distance determines the total distance (in meters) traveled by a car during the braking time interval. The correct state can be defined if the final vehicle velocity (FV) is less than a specified value *fv* (m/sec) and the stopping distance (SD) is less than *sd* (meters). The relative fault severity levels can also be introduced using some knowledge on car behavior e.g. a car with a greater final velocity has a greater final momentum and thus will likely cause more damage during a head-on collision between two vehicles. The acceptable values for SD and FV can be based on the analysis given in [18,19] where the authors specify nominal values of stopping distance for a car traveling at a given initial velocity. This approach is illustrated in the next section.

The performed experiments were targeted at transient faults (bit flips) injected into registers (specified CPU or FPU registers, or all of them), the code or data area of the memory used by the ABS controller. By concentrating on specified system resources (or code segments) we can perform a deeper analysis and tune appropriate fault handling mechanisms. For each experiment, we choose a representative set of input data to assure high coverage of the code, decisions etc. The number of injected faults is sufficiently large to assure statistical significance of the obtained results.

In many applications fail-silent hypothesis is assumed i.e. the system produces correct outputs and stops producing outputs after detecting an error. For systems with some inertia as well with control loops fail-bounded hypothesis can also be assumed [17]. In this case the controller produces correct outputs, does not produce outputs after detecting some errors, produces wrong outputs within an acceptable deviation margin from the correct ones. This margin can be defined by some application dependant assertions. This approach allows postponing or even eliminating recovery. Here the assertion should allow predicting critical situations. Within the idle time of a controller we have to check if at the present control trajectory the worst case error may produce a critical situation. If so, we have to perform recovery. In the opposite case, the system follows its normal operation. Recovery can be limited to software (re-execution of some procedures, etc.) or, if needed, instantiating spare or backup hardware resources can do it.

4 Experimental Results

All simulations were performed with FITS injector within IBM PC platform (XP Windows Professional) and the model of the environment. The ABS controller and its environment model constitute an integrated program CE written in C++ language. This program was implemented based on mathematical models from [18] (compare section 2). Integrating these models we assure that the used variables and codes are disjoint. Hence, disturbing the ABS controller we do not interfere with environment model. The initial conditions of fault injection experiments are defined by setting some parameters in CE e.g. initial car speed, mechanical characteristics, road conditions (compare section 2). Each experiment is composed of a specified number of tests (a single fault injection) which are performed according to the predefined scenario (e.g. pseudorandom injections into code with static or dynamic strategy). For each test the results (system behavior) are stored in a file for the purpose of detailed analysis. We have also developed a special result qualification module (RQ) which analyzes the system behavior and identifies correct or incorrect system status. This status is based on the predefined criteria described in section 3. The qualification decision is sent to the fault injector FITS which accumulates statistics from all the tests. Moreover, it identifies timeouts and system exceptions.

The ABS control program was based on the model described in section 2. We have considered two implementations: the basic version (BV) and fault hardened versions (VH1, VH2). The basic version is a direct implementation of the mathematical model from [18,19]. In the fault hardened version we use the built in hardware and software fault detection mechanisms, which generate system exceptions. Typically these exceptions are signaled by the operating systems. It is possible to take over most of them at the application level and perform some error recovery. For this purpose we can use the *try* and *catch* construct provided by object oriented languages.

It assures taking over exceptions (specified by the filter or all of them – *catch(...)*) generated during the execution of the code within the *try* brackets (*try {segment of the application code}*). For any specified exception (in the exception filter) we define an appropriate handling procedure. For example, it may initiate reexecution of the program code starting from some specified checkpoint (previously established – backward recovery), suspending further execution of the thread, etc. The error correction can be made, if the error is uniquely correlated with the disturbed program segment. In version VH1 each captured exception initiates floating point unit (FPU) reset using the *_fpreset* function. In version VH2 additional code and state recovery is added by using a redundant DLL library and loading its static copy of the microcontroller code whenever a system exception was raised.

The size of the generated MSVC 2005 compiler binary image of the ABS controller program is approximately 100 KB for the basic version, and only 10-15 KB larger for versions with fault tolerance mechanisms VH1 and VH2. The ABS microcontroller without fault tolerance mechanisms consists of about 640 static code assembly instructions and 6 million dynamic instructions executed within the 15,000 simulation iterations. The source code was about 2000 lines in C++. The tire, wheel and suspension environment model consists of about 1000 static code instructions and 12 million dynamic instructions (for the analyzed test scenario).

In the discussed experiments, the operation of ABS is simulated for a fast braking scenario (fast pressing of the brake pedal). The braking process is performed in response to this input signal, taking into account the wheel speed, acceleration, etc. Observing the system behavior we can monitor internal variables or output control signals. The most interesting signals relate to the brake torque, car horizontal velocity and stop distance. The presented results relate to the car initial speed 60km/hr and a dry road. The golden run trajectories of these signals for the considered test scenario are given in Fig. 2a, 2b and 2c (they cover real ABS operation time of 1.5 s). The brake signal (pressing brake pedal by the driver) activates the ABS controller till the moment of achieving velocity 1.5 m/s. So the braking distance is 14.5m at this moment which corresponds to 1.5 sec time. For comparison, we give in fig. 2d-f the plots of the same variables in the case of an injected fault at time moment t=0.5438s (random bit-flip in an instruction code of the ABS controller). The plots differ

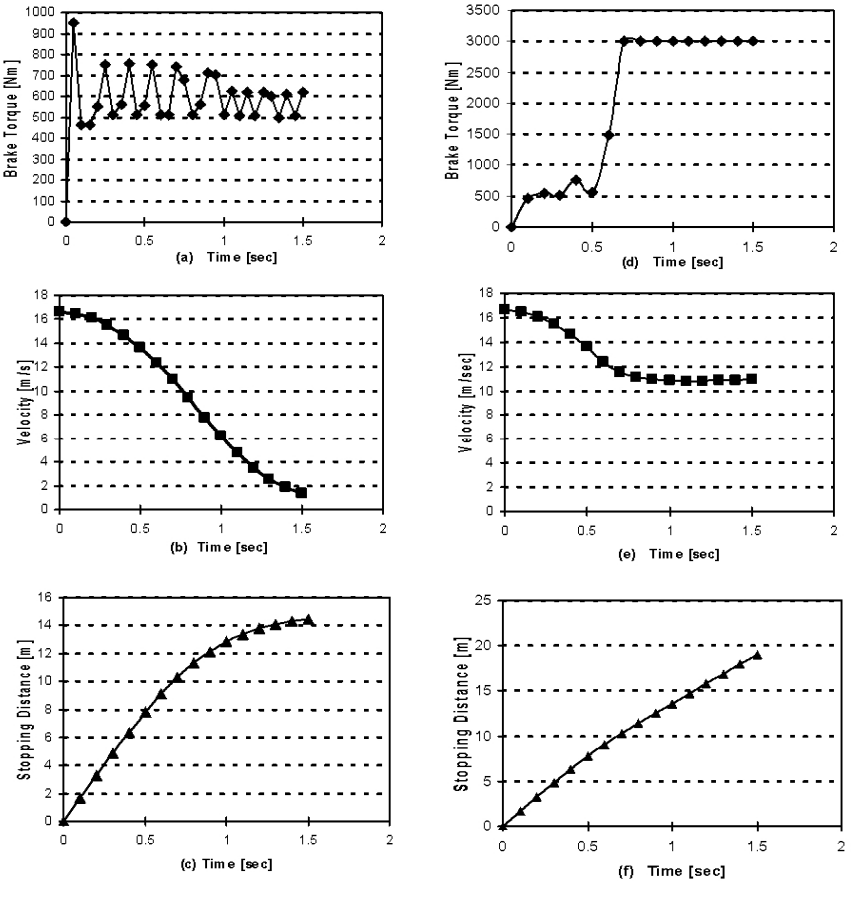

Fig. 2. Sample plots showing ABS braking: *a*), *b*) and *c*) golden run plots for the brake torque, vehicle speed and traveled distance in time, respectively; *d*), *e*) and *f*) corresponding plots related to an injected bit-flip fault in ABS code at t=0.5438s

significantly from the golden run; in particular, the braking distance at time 1.5 sec is 19 m and the final speed is 11m/s. This corresponds to a dangerous situation. Such analysis can be done manually for some selected faults to get knowledge of their impact.

More interesting are statistical results over many faults. We have injected many faults into the ABS controller code, CPU or FPU registers and data memory. We have assumed that the correct behavior corresponds to the final speed FV < 1.5 m/s at t=1.5 sec and stopping distance SD < 16 m. This criteria is easier to calculate than checking the correctness of the brake torque trajectory.

Test results for the basic controller version are shown in Fig. 3. Fault locations REG, MEM, FPU and INSTR correspond to CPU registers, data memory area, FPU registers and memory code area, respectively. The fault triggering moments are generated pseudo-randomly and distributed equally in time (dynamic). For the location CODE faults are injected with equal distribution in the memory code area space (static). A large percentage of faults resulted in system exceptions, which were not handled, and in fact they lead to dangerous situations. Relatively small percentage of incorrect results is due to some natural fault tolerance of the used control algorithm. A simple recovery based on taking over exceptions increase significantly the correct result percentage (Fig. 4). The presented results relate to latched transient faults (bit flips in registers or code/data memory). For comparison we give results of fault injections into the code for non-latched transient faults in the ABS controller memory.

This mimics the controller implementation with code stored in a non-volatile memory e.g. flash. For version VH1 we obtained INC=1%, C=94% and S=5%. This confirms significantly lower fault susceptibility for non-latched faults than for latched (compare Fig. 3). Similar results can be achieved for latched faults if more efficient error recovery mechanisms are employed (e.g. those discussed in our previous paper [9]).

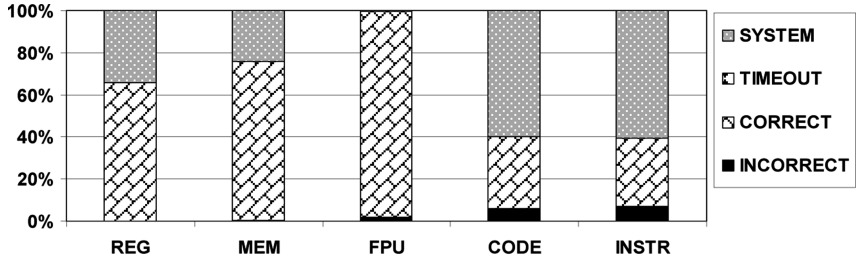

Fig. 3. Test results for the ABS micro-controller basic version (*BV*)

We have also developed the controller version VH2* adapted to platform x86 but without floating point unit (FPU). In this case all floating point calculations are done in software. The number of executed instructions for the analyzed test scenario increased from about 14 million in version VH2 to over 100 million in version VH2*. Fault susceptibility of both versions was practically the same for faults injected into registers and data memory. Faults injected into the code memory gave more correct results for version VH2* (84.5%) than for VH2 (74%).

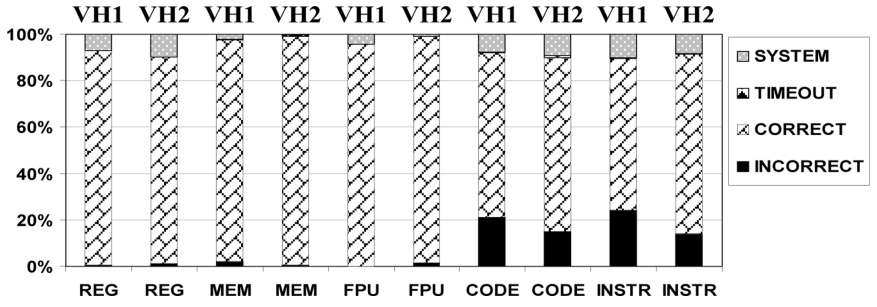

Fig. 4. Test results for the ABS with improved fault tolerance: version *VH1* and *VH2*

5 Conclusion

The main goal of this paper was to verify the developed methodology for evaluating the impact of faults on reactive systems. We have adapted our fault injector tool (FITS) for such systems by integrating it with the analyzed application (microcontroller) and its environment. Moreover, we added an interface to deal with specialized result qualification module and test scenario configuration. This approach has been successfully verified for the real ABS microcontroller and practical test scenarios. The proposed approach allows detailed behavioral analysis of the system in the presence of faults and gives statistics on susceptibility to faults injected in specified circuit areas. This approach is very useful in finding fault leakage sources, optimization of fault handling procedures as well as evaluation of the final projects. It was also verified for other applications [16]. We can identify program modules and data which are the most sensitive to faults, analyze critical behavior of the system in the presence of faults, etc. We can also evaluate the effectiveness of embedded fault handling procedures.

As opposed to classical calculation oriented applications, real-time and reactive systems require more complex result qualification methods. They can be based on observing output control signals or selected parameters describing the quality of performed tasks. This approach seems to be more effective and this was proved for the ABS controller. The experimental results showed that the basic version of the ABS controller comprises some natural fault tolerance capabilities (due to the used algorithm). This can be improved with simple exception handling procedures as well as by using non-volatile memory for the code. In the performed experiments we use a model of the control object and the system environment. Hence, the results depend upon the accuracy of the used models. This drawback can be eliminated in experiments with real objects, but such experiments are usually too expensive. Moreover there is a danger of causing critical situations or some damages in the case of injected faults.

Further research is targeted at developing and analyzing more effective fault tolerance mechanisms. Here we plan to use our experience gained with calculation oriented applications [9] and apply it to reactive systems. Result qualification will be extended by introducing more categories, e.g. loss of braking (the wheel speed is zero for some specified minimal time), locked wheel (the wheel speed does not decrease for more than some specified time). Moreover, we will consider distributed systems e.g. around a CAN network [10,15].

Acknowledgment. This work was supported by Ministry of Science and Higher Education grant 4297B/T02/2007/33.

References

1. Corno, F., Esposito, E., Reorda, M., Tosato, S.: Evaluating the effects of transient faults on vehicle dynamic performance in automotive systems. In: ITC 2004, pp. 1332–1339. IEEE Press, Los Alamitos (2004)
2. Dilger, E., Karrelmeyer, R., Straube, B.: Fault tolerant mechatronics. In: IOLTS 2004, pp. 214–218. IEEE Press, Los Alamitos (2004)
3. Mariani, R., Fuhrmann, P., Vittorelli, B.: Fault Robust Microcontrollers for Automotive Applications. In: IEEE On-line Test Symposium, pp. 213–218. IEEE Press, Los Alamitos (2006)
4. Gaid, M., Cela, A., Diallo, S.: Performance Evaluation of the Distributed Implementation of a Car Suspension System. In: PDS 2006. IFAC Press (2006)
5. Nouillant, F., Aisadian, X., Moreau, A., Oustaloup, et al.: Cooperative Control for Car Suspension and Brake Systems. J. of Auto. Tech. 4(4), 147–155 (2002)
6. Zalewski, J., Trawczynski, D., Sosnowski, J., Kornecki, A., Sniezek, M.: Safety Issues in Avionics and Automotive Databuses. In: IFAC World Congress. IFAC Press (2005)
7. CEI International standard IEC 61508 (1998-2000)
8. AUTOSAR partnership, http://www.autosar.org
9. Gawkowski, P., Sosnowski, J.: Experimental Evaluation of Fault Handling Mechanisms. In: Voges, U. (ed.) SAFECOMP 2001. LNCS, vol. 2187, pp. 109–118. Springer, Heidelberg (2001)
10. Short, M., Pont, M.J.: Fault tolerant time-triggered communication using CAN. IEEE Transactions on Industrial Informatics 3(2), 131–142 (2007)
11. Adermaj, A.: Slightly-of-specification failures in the time triggered architecture. In: 7th IEEE Int. Workshop on High Level Design and Validation and Test, pp. 7–12. IEEE Press, Los Alamitos (2002)
12. Anghel, L., Leveugle, R., Vanhauwaert, P.: Evaluation of SET and SEU effects at multiple abstraction levels. In: 11-th IEEE IOLTS Symposium, pp. 309–314. IEEE Press, Los Alamitos (2005)
13. Arlat, J., Crouzet, Y., Karlsson, J., Folkesson, P., Fuchs, E., Leber, G.H.: Comparison of physical and software implemented fault injection techniques. IEEE Transactions on Computers 52(9), 1115–1133 (2003)
14. Cervin, A., Henriksson, D., Lincoln, D., Eker, J., Årzén, K.: How Does Control Timing Affect Performance? IEEE Control Systems Magazine 23(3), 16–30 (2003)
15. Trawczynski, D., Sosnowski, J., Zalewski, J.: A Tool for Databus Safety Analysis Using Fault Injection. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 261–275. Springer, Heidelberg (2006)
16. Gawkowski, P., et al.: Software Implementation of Explicit DMC Algorithm with Improved Dependability. In: Int. Joint Conf. on Computer, Information, and Systems Sciences, and Engineering (CISSE 2007), December 3 - 12 (2007)
17. Cunha, J., Rela, M., Silva, J.: On the Use of Disaster Prediction for Failure Tolerance in Feedback Control Systems. In: Dependable Systems and Networks 2002, pp. 123–134. IEEE Press, Los Alamitos (2002)
18. Rangelov, K.: Simulink Model of a Quarter-Vehicle with an Anti-Lock Braking System. Research Report, Eindhoven University of Technology (2004)
19. MSC Software: Using ADAMS/Tire. ADAMS Software Manual (2005)