Fault-Injection Tool for Distributed Elevator System

Branislav D. Petrovic[1], Goran S. Nikolic[2]

Abstract – **In this paper, a *Fault Injection Environment* (*FIE*) for distributed elevator system (*DES*) is presented. The *FIE* is suitable to assess the correctness of the design and implementation of the hardware and software mechanisms existing in embedded microprocessor-based systems, and to compute the fault coverage they provide. The paper describes and analyzes different solutions for implementing the most critical modules. In addition, a powerful technique for emulating hardware faults is developed. Having in mind that our embedded system is hierarchical type, very important segment is communications. A safety-critical system needs fault-tolerant communication between its components. This is especially important for distributed real-time systems that are based on the results of the communication. In addition to, having in mind that the *FIE* implementation is used in lift processor of distributed structure a number of experimental runs in relatively short time can be executed. On the other hand, a number of faults were injected in simulation model of prototype implementation and system behaviour, in real life environment, is observed.**

Keywords – **Fault Injection Tools, Fault Tolerant System, SWIFIT Model, Lift (Elevator) system.**

I. Introduction

Modern technological systems rely heavily on sophisticated control systems to meet increased safety and performance requirements. This is particularly true in safety critical applications, such as aircraft, spacecraft, nuclear power plants, and chemical plants processing hazardous materials, where a minor and often benign fault could potentially develop into catastrophic events if left unattended for or incorrectly responded to. To prevent fault-induced losses and to minimize the potential risks, new control techniques and design approaches need to be developed to cope with system component malfunctions whilst maintaining the desirable degree of overall system stability and performance levels. A control system that possesses such a capability is often known as a *FTCS* (*Fault-Tolerant Control System*). Fault tolerance is the ability of a system to perform its function correctly even in the presence of internal faults. The purpose of fault tolerance is to increase the dependability of a system. A complementary but separate approach to increasing dependability is fault prevention. This consists of techniques, such as inspection, whose intent is to eliminate the circumstances by which faults arise. To increase system dependability we use in general three techniques: fault avoidance, fault masking and fault tolerance. [1]

1Branislav D. Petrovic is with the Faculty of Electronic Engineering, University of Nis, Aleksandra Medvedeva 14, 18000 Nis, Serbia, E-mail: branislav.petrovic@elfak.ni.ac.rs

2Goran S. Nikolic is with the Faculty of Electronic Engineering, University of Nis, Aleksandra Medvedeva 14, 18000 Nis, Serbia, E-mail: goran.nikolic@elfak.ni.ac.rs

The main idea of fault avoidance techniques is to prevent fault occurrence. This is achieved by design reviews and automation, part selection, screening, lowering power consumption, software rejuvenation etc. Fault masking techniques hide the faults and prevent occurrence of errors using error correction codes or passive redundancy e.g. triple modular redundancy with voting. Fault tolerance techniques detect faults, identify them and perform appropriate recovery (e.g. replacing a faulty model by a spare one).

One of the most used digital systems, today, is microprocessor-based embedded systems. Fault tolerance mechanisms, in this case, are introduced at the hardware and software level. Debugging and verifying the correct design and implementation of these mechanisms ask for effective environments, and Fault Injection represents an acceptable solution for their implementation.

Fault tolerance and reliability measures cannot be evaluated using benchmark programs and standard test methodologies, but only by observing the system behaviour when a fault appears inside it. Since the MTBF (*Mean Time Between Failure*) in a safety-critical system can be of the order of years, fault occurrence has to be artificially accelerated in order to observe the system behaviour under faults without waiting for the natural appearance of actual faults.

On the other hand, study of failures and errors is an important part in the evaluation of system reliability. To understand the potential failures, there have been developed experimental techniques that can be applied both to the hardware and to the software. These techniques not only are suitable during the phase of system analysis and design, but also during their prototyping and manufacturing phases, in other words during the whole of system life cycle.

II. Faults, Errors, Failure

A. Definition and Examples

Implicit in the definition of fault tolerance is the assumption that there is a specification of what constitutes correct behaviour. A failure occurs when an actual running system deviates from this specified behaviour. The cause of a failure is called an error. An error represents an invalid system state, one that is not allowed by the system behaviour specification. The error itself is the result of a defect in the system or fault. In other words, a fault is the root cause of a failure. That means that an error is merely the symptom of a fault. A fault may not necessarily result in an error, but the same fault may result in multiple errors. Similarly, a single error may lead to multiple failures. [2].

A number of hazardous faults that can lead to accident are obvious in plenty of elevators. To explain most common faults in detail, let consider one standard traction elevator.

Traction machines are driven by AC or DC electric motors. The machines use gears to mechanically control movement of elevator cars by "rolling" steel hoist ropes over a drive sheave which is attached to a gearbox driven by a high speed motor. These machines are generally the best option for basement or overhead traction use for speeds up to 5 m/s. A brake is mounted between the motor and drive sheave (or gearbox) to hold the elevator stationary at a floor. This brake is usually an external drum type and is actuated by spring force and held open electrically; a power failure will cause the brake to hold the elevator in position. In each case, cables are attached to a hitch plate on top of the cab, and then looped over the drive sheave to a counterweight attached to the opposite end of the cables which reduces the amount of power needed to move the cab. The counterweight is located in the hoist-way and rides a separate rail system; as the car goes up, the counterweight goes down, and vice versa. This action is powered by the traction machine which is directed by the controller, typically a relay logic or computerized device that directs starting, acceleration, deceleration and stopping of the elevator cab.

Let describe the first hazard. Suppose the situation: lift car is at one of bottom floors and is about to start moving up. At the moment of starting the brake is energized and motor is powered on. If cabin is fully loaded, the start up current can be of high intensity. In this case a fuse failure or contactor failure can disconnect the motor from power supply, but brake is still energized. Detection of power failure at the motor leads is of great importance. Measuring of voltages is not sufficient because of electromagnetic induction in motor windings. If controller is not capable to detect the failure, cabin will go to move uncontrolled, probably falling down to dampening device. The situation obviously leads to accident injuries.

Second hazard is related to elevator doors opening to an open shaft. If elevator door is opened but cabin is not on that floor, the lift controller mast immediately stops the car moving. If door opening sensor is shortened for some reasons, the stopping function cannot be achieved. Besides, a door opened with no cab on the floor is dangerous situation and mast be obviously signalized.

Next hazard that can cause accident is erroneous determination of the cabin position. In old fashion elevators, the position of the car is determined by counting method. Passing the cab near some kind of proximity sensor on every floor a counting floor register in controller is incremented or decremented. In the case of transient error or proximity sensor malfunction, determination of the position is not possible. New solutions of lift controllers that use some kind of encoders are also prone to transient errors and power supply interruption. Hence, accident situation can occur if an outermost floor is missed.

All of above hazardous situations must be predicted lift controller response proved. Using the *FIT* verification of controller response can be achieved effectively.

B. Faults Classification

It is helpful to classify faults in a number of different ways, as shown by the UML class diagram in Fig. 1.

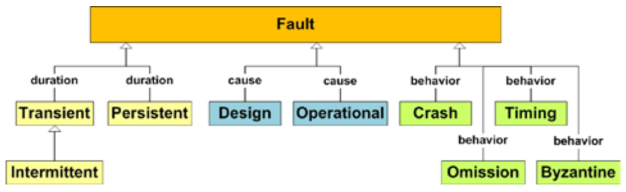


Fig. 1.Different Classifications of Faults

Based on *duration*, faults can be classified as *transient* or *permanent*. A transient fault will eventually disappear without any apparent intervention, whereas a permanent one will remain unless it is removed by some external agency. A different way to classify faults is by their underlying *cause*. *Design faults* are the result of design failures. *Operational faults*, on the other hand, are faults that occur during the lifetime of the system and are invariably due to physical causes, such as processor failures or disk crashes. Finally, based on how a failed component behaves once it has failed, faults can be classified into the following categories: 1. *Crash faults* -- the component either completely stops operating or never returns to a valid state; 2. *Omission faults* -- the component completely fails to perform its service; 3. *Timing faults* -- the component does not complete its service on time; 4. *Byzantine faults* -- these are faults of an arbitrary nature.

C. Types of Redundancy a Fault Tolerance

All of fault tolerance is an exercise in exploiting and managing *redundancy*. Redundancy is the property of having more of a resource than is minimally necessary to do the job at hand. As failures happen, redundancy is exploited to mask or otherwise work around these failures, thus maintaining the desired level of functionality. There are four forms of redundancy that we will study: hardware, software, information, and time.

Hardware faults are usually dealt with by using hardware, information, or time redundancy, whereas software faults are protected against by software redundancy. Hardware redundancy is provided by incorporating extra hardware into the design to either detect or override the effects of a failed component. The best-known form of information redundancy is error detection and correction coding. Here, extra bits (called check bits) are added to the original data bits so that an error in the data bits can be detected or even corrected. Note that these error codes (like any other form of information redundancy) require extra hardware to process the redundant data (the check bits). Error-detecting and error-correcting codes are also used to protect data communicated over noisy channels, which are channels that are subject to many transient failures.

Time redundancy can thus be used to detect transient faults in situations in which such faults may otherwise go undetected. Time redundancy can also be used when other means for detecting errors are in place and the system is capable of recovering from the effects of the fault and repeating the computation. Software redundancy is used mainly against software failures. one way is to independently produce two or more versions of that software Just as for hardware redundancy, the multiple versions of the program can be executed either concurrently (requiring redundant hardware as well) or sequentially (requiring extra time, i.e., time redundancy) upon a failure detection.

D. Basic Measures of Fault Tolerance

A measure is a mathematical abstraction that expresses some relevant facet of the performance of its object. By its very nature, a measure only captures some subset of the properties of an object. The goal in defining a suitable measure is to keep this subset large enough so that behaviours of interest to the user are captured, and yet not so large that the measure loses focus.

a) *Traditional Measure*

Two of these measures are reliability and availability. Closely related to reliability is the *Mean Time to Failure*, denoted by *MTTF*, and the *Mean Time Between Failures*, *MTBF*. The first is the average time the system operates until a failure occurs, whereas the second is the average time between two consecutive failures. The difference between the two is due to the time needed to repair the system following the first failure.

b) *Network Measure*

There are more specialized measures, focusing on the network that connects the processors together. The simplest of these are classical node and line connectivity, which are defined as the minimum number of nodes and lines, respectively that have to fail before the network becomes disconnected.

III. SOFTWARE IMPLEMENTED HARDWARE FAULT INJECTION TECHNIQUE

The general approach is to treat reliability as a system problem and to decompose the system into a hierarchy of related subsystems or components. The reliability of the lift system is related to the reliability of the hardware, software and human components. Software development is quite different from hardware development because software reliability is based on the varying values of the input, the huge number of input cases, the initial system states, and the impossibility of exhaustive testing. On the other hand, source of most hardware errors is equipment failure.

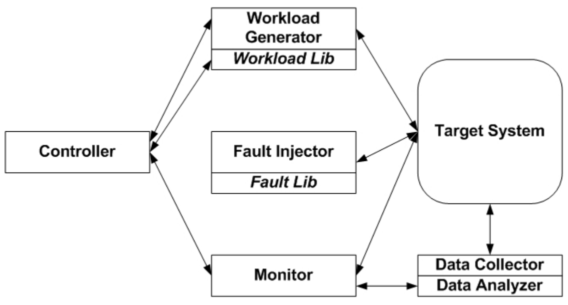


Fig. 2. Basic Components of a Fault Injection Environment

Mechanical hardware can jam, break, and become worn-out, and electrical hardware can burn out, leaving open or short circuit etc. Software Implemented Fault Injection Technique (*SWIFIT*) is a very attractive since it does not need specific hardware to realize the fault injection (Fig. 2.).

It can be used to prove the failure tolerance mechanisms at different levels of system abstraction including architectural, functional, logical, and electrical and allows us the control of the location, time, duration, and type of the injected faults much more easily than does physical injection.

A global functional structure of the lift processor system is given in Fig.3. Structure is composed of a number of lift processor clusters *LPCi*. The system is intended for controlling more lift units so-called multiplex lift system (duplex, triplex). The clusters are connected by *XNET* bus based on *RS485*. As can be seen on Fig.4, the one lift processor cluster is of distributed structure connected by *LNET* bus, also of *RS485* type. The lift processor cluster is composed of following nodes: Master node, M, which directly controls most of actuators in system (motor, valves, brakes, and others). Cabin node, *CAB*, acquire all information from moving car, and from automatic door control. Register box, *RB*, for collecting requests from passengers in lift and displaying all necessary information. A corresponding number of floor processors *FPi* on each floor. Getaway for connecting to *XNET* bus is realized trough master node.

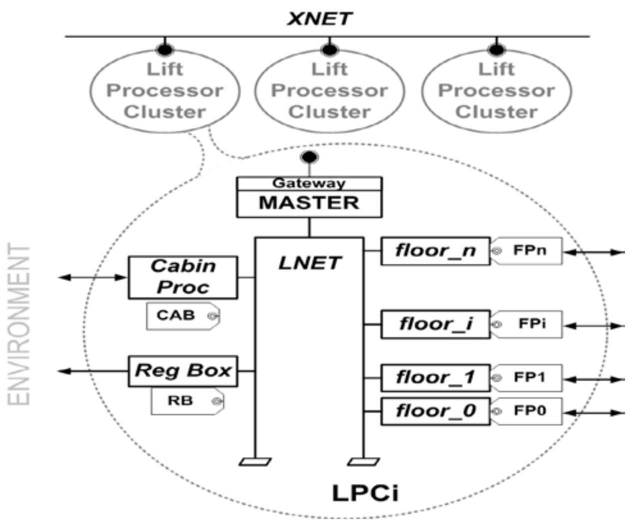


Fig. 3. Topology lift system

This approach would provide the desired flexibility, and at the same time, would allow us to execute many experimental runs in a relatively short time. The generally accepted solution to this problem is to inject the faults in a simulation model or a prototype implementation [4], and to observe the behaviour of the system under the injected faults.

Previously mentioned flexibility is consequence of topology system i.e. distributed pattern. This solution of the communication segment gives us possibility to insert the fault injection tool between some communication node and in this way simulates every possible hardware faults. The structure gives us possibility to insert the fault injection tool any point in *LNET* or *XNET* bus. Hence, in this way we can simulate most of hardware faults. Additionally, the *SWIFIT* can be used in development phase as simulation tool. Model that is used in this approach in the literature is well known like the *FARM*

Model. The major requirements and the problem related to the development and application of a validation methodology based fault injection are presented through *FARM* model. When the fault injection technique is applied to a target system, the input domain corresponding to a set of faults *F*, and a set of activation *A*, which consist of a set of test data patterns aimed at exercising the injected faults, and the output domain corresponds to a set of readouts *R*, and set of derived measures *M*.

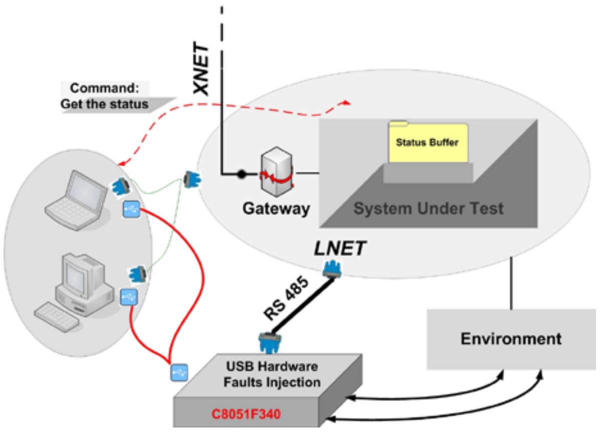
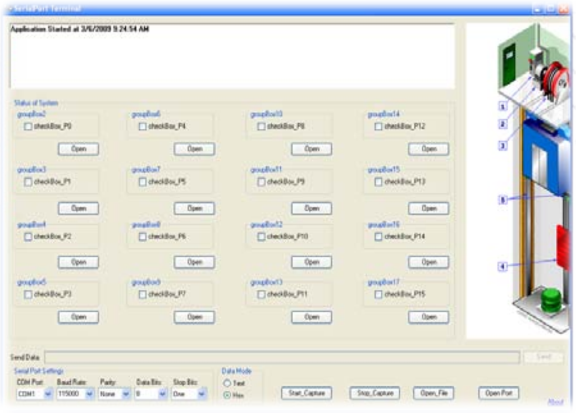


Fig. 4. Operation mode of *SWIFIT* model

In this paper, we look output domain measurement segment using the *PC* oriented *Graphical User Interface* environment that is written in Dot Net C# language. Screenshot of this program is give on the Fig. 5.

The resulting measures are computed from the data that is collected by the system in the several test-runs (*Status_Buffer*) and at the request of the user forward to PC where are stored in so-called dump file.



Application Started at 3/6/2009 9:24:54 AM

Serial Port Settings: COM Port: COM1, Baud Rate: 115000, Parity: None, Data Bits: 8, Stop Bits: One

Data Mode: Text [✓ Hex]

Fig. 5. Screenshot of the program

Measurement is an off-line process carried out in function of the objective of the fault injection campaign. It is also important the validation of the target system that guarantees its correct specified behavioural in a failure scenario, in the case that fault and error propagates to the output. In addition to the above-mentioned off line measurement and analysis, program has on-line on the visual presentation of error detection and locating sources of error activating image sensor in the flash presentation of the program.

This approach also tested the software in two ways:

1. Logical software bug across on-line debugging method and,
2. Recovery code - the part of the code that is designed to respond in the case to detect error states.

Recovery code should gracefully restore the system to a valid state before a failure occurs. Our goal is to create potentially error free or zero-probability software with creating fault tolerant software, and to demonstrate that a software is completely correct for many number of possible executions scenario in the real life.

This approach also provides an opportunity to developed software, after the phase of the system design and analysis, can be applied to remote tracking and monitoring of the system status with minor hardware redundancy interface (*Ethernet Controller* or *WLAN 802.11* modules for Internet access, or *GSM* module).

IV. Conclusion

In this paper part of the *FARM* model is presented. Our goal has been to make the validation of the target system that guarantees its correct specified behavioural in a failure scenario and to understand the processes in the segment of communication, timing and protocol on the physical and the higher levels of ISO/OSI model. Hese actions consist of detecting the fault, identifying the system component affected by the fault, and taking an appropriate recovery action which may involve system reconfiguration. Each of these actions takes time that is not a constant but may change from one fault to another and may also depend on the current workload.

Also, should be noted that in the paper mentioned some of the possible scenarios and solutions that are arising from the analysis. In this way, there has been a major tendency to develop reliable software for a very complex system such as the lift with which it is possible to lower price hardware implementation. During the exploitation of proposal model, system components whose failure is more likely to result in a total system crash are identified, also identified optimal chekpointing having in mind that it is a distributed system, developed a reliable protocol, considered problem the synchronization at the level of the system, considered the scenario a violent intrusion into the system, etc. Mention that the software solution and has RS232 and USB interface to *SUT*, and that in this way achieved universality and flexibility of the program that makes it suitable for other application

References

[1] I. Koren, C. Mani Krishna,, *Fault tolerant system*, San Francisco, Elsevier, Inc., 2007

[2] B. Selic, “Fault tolerance techniques for distributed systems”, www.ibm.com

[3] M. L. Shooman, "RELIABILITY OF COMPUTER SYSTEMS AND NETWORKS Fault Tolerance, Analysis, and Design", New York, John Wiley & Sons, Inc., 2002.

[4] Branislav D. Petrović, Goran S. Nikolić, Koncepcija procesora lifta sa distribuiranom strukturom, ZBORNIK RADOVA 31.KONGRESA SA MEDJUNARODNIM UČEŠĆEM – HIPNEF 2008, Vrnjačka Banja, Srbija, 15 – 17. oktobar 2008, pp. 179 – 184