SCIENTIFIC PUBLICATIONS OF THE STATE UNIVERSITY OF NOVI PAZAR
SER. A: APPL. MATH. INFORM. AND MECH. vol. 3, 1 (2011), 43-51.

# Software Tool for Distributed Elevator Systems

**B. D. Petrović, G. S. Nikolić**

**Abstract:** In this paper software development and testing tool for distributed elevator systems (DES) is described. The motivation for making this tool grew out of the problems that have arisen in the process of designing, testing and servicing of the elevator control processor (referred to as lift-processor), and the entire elevator system. The problems are largely a consequence of the impossibility of viewing all the functional requirements or changes to these requirements during the design phase or exploitation. It should be noted that the lift-processor must meet the basic requirement that is placed in front of such a system, related to the high security of the whole system. In fact, despite the availability of a wide range of tools for designing reliable hardware and software, a lack of such tools that closely covers specific problems related to the proposed lift-processor architecture is evident. Also, this device could achieve much faster response to the new functional demands of the market.

**Keywords:** Fault tolerant, fault injection, elevator systems

## 1 Introduction

Computer-based systems have increased dramatically in scope, complexity, and pervasiveness, particularly in the last few years. Most industries are highly dependent on computers for their basic day-to-day functioning. Safe and reliable software operation is a significant requirement for many types of systems, for instance, in aircraft and air traffic control, medical devices, nuclear safety, petro-chemical control, high-speed rail, electronic banking and commerce, automated manufacturing, military and nautical systems, for aeronautics and space missions, and for appli-ance-type applications such as automobiles, washing machines, temperature control, and telephony, to name a few. The cost and consequences of these systems failing can range from mildly annoying to catastrophic, with serious injury occurring or lives lost, systems (both human-made and natural) destroyed, security breached, businesses failed, or opportunities lost [1].

As software assumes more of the responsibility for pro-viding functionality in these systems, it becomes more complex and more significant to the overall system per-formance and dependability.

Manuscript received October 21 2010 ; revised December 25 2010; accepted March 1 2011
B. D. Petrović, G. S. Nikolić are with the Faculty of Electronic Engineering, University of Niš, Serbia.

Ideally, the processes by which software is conceptual-ized, created, analyzed, and tested would have advanced to the point where software could be developed without errors.

The current state-of-the-practice is such that fewer errors are introduced, but unfortunately not all errors are pre-vented. Even if the best people, practices, and tools are used, it would be very risky to assume the software devel-oped is error-free. There may also be cases in which an error, found late in the system's life cycle and perhaps prohibitively expensive to repair, is knowingly allowed to remain in the system.

### 1.1 Generally about Fault Tolerant Systems

To increase system dependability we use in general three techniques: fault avoidance, fault masking and fault tolerance. The main idea of fault avoidance techniques is to prevent fault occurrence. 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. re-placing a faulty model by a spare one).

The propagation of a fault to an observable failure follows a well defined cycle. When executed, a fault may cause an error, which is an invalid state within a system boundary. An error may cause further errors within the system boundary, therefore each new error acts as a fault, or it may propagate to the system boundary and be observable. When error states are observed at the system boundary they are termed failures. This mechanism is termed the fault-error-failure cycle and is a key mechanism in depend-ability.

The other hand, fault tolerance is a feature of the system to continue implementation of its activities to the same or reduced level of performance, but in any case in a way that is safe for the environment, to appear in spite of unexpected hardware or software errors. In essence, the goal of fault tolerance is to protect the system from the propagation of errors to its output, and therefore failure.

Fault tolerance and reliability measures cannot be evaluated using benchmark programs and standard test method-ologies, but only by observing the system behavior when a fault appears inside it. Since the *MTTF* (*Mean Time To Failure*) and *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 behavior under faults without waiting for the natural appearance of actual faults.

### 1.2 Levels of Abstraction of Fault Injection

First, we consider fault injection technique. Therefore, fault injection technique is the process of intention-ally incorporating errors in code in order to measure the ability (of the tester, or processes) to detect such errors [2]. Various levels of abstraction may be identified for the application of fault injection, depending on the type of model used for the target system. Three major types of models are: axiomatic, empirical and physical models [3].

A prototype of real fault injector is of the interest. For the purposes of adequate testing of the elevators system (lifts) the following two types of simulations, are used:

- Software implemented fault injection
- Simulation with hardware in the loop

Simulation with hardware in the loop is a form of simulation in real time. In contrast to pure simulation in real-time, simulation with hardware in the loop have additional real components in the loop. The components can be, for example, an electronic control module or a real engine. In both cases we inject faults into the system to

- identify dependability bottlenecks,
- study system behavior in the presence of faults,
- determine the coverage of error detection and recovery mechanisms, and
- evaluate the effectiveness of fault tolerance mechanisms (such as reconfiguration schemes) and performance loss.

Faults are injected either at the hardware level (logical or electrical faults) or at the software level (code or data corruption) and the effects are monitored.

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 elevator (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. Mechanical hardware can jam, break, and become worn-out, and electrical hardware can burn out, leaving open or short circuit etc. The system used for evaluation can be either a prototype or a fully operational system. Injecting faults into an operational system can provide information about the failure process [4].

## 2 Fault injections tool

### 2.1 Description of the Tool

The fault injection tool, named ALIEN, is structured well defined five subsystems:

- Activator: activates the target system, allowing it to be tested in its normal conditions.
- Injector: does the injection of the faults inside the sys-tem under test.
- Monitor: monitors the target system, in order to verify expected operating conditions.
- Controller: controls the subsystems above, so they do their activities coordinately.

- User interface: receives the specifications from the user for the experiment execution and gives back the results.

Of the subsystems above, only injector, monitor and activator communicate with the system under test (*SUT*). They exchange messages using the controller subsystem. The controller will receive experiment specifications from the user interface, in form of the parameters that will be passed to the subsystems. These, afterwards, will send back a part of data obtained in the execution of the experiment. Rest of the data will be stored in the *SUT* and, through hardware monitor interface, will be passed back to the user interface. The diagram in Fig 1 shows the structure of the proposed.

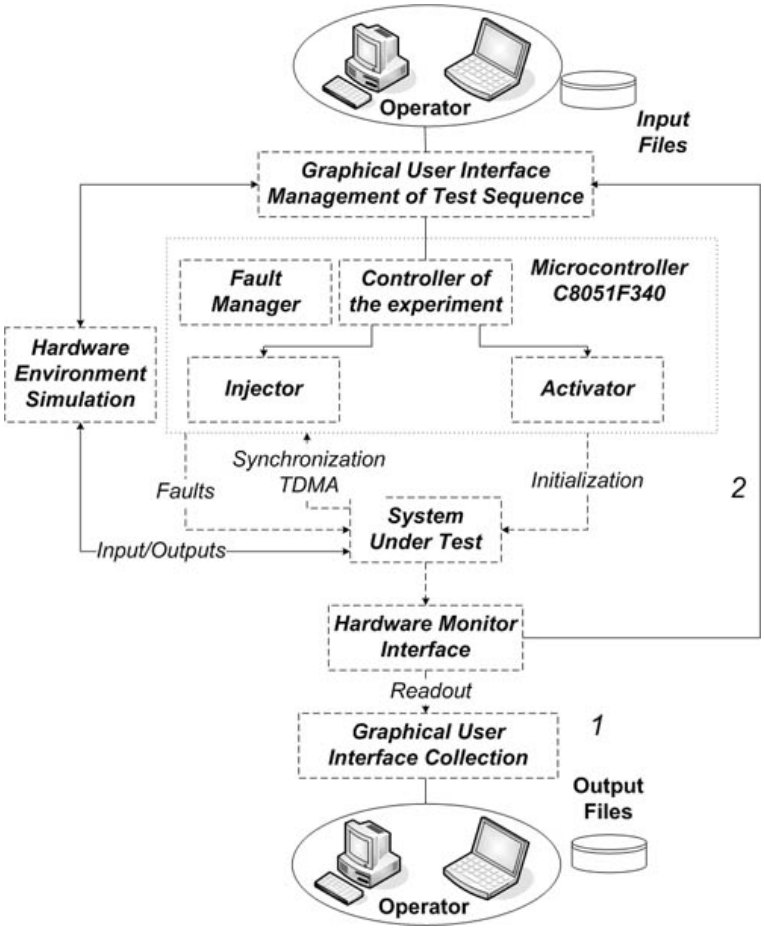


Fig. 1. Fault Injector architectural pattern structure

## 2.2 Architecture of DES

A global functional structure of the lift processor system is given in Fig. 2. Structure is composed of a number of lift processor clusters *LPCi*. As 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 2, the lift processor cluster has distributed structure. Nodes are 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 for 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.

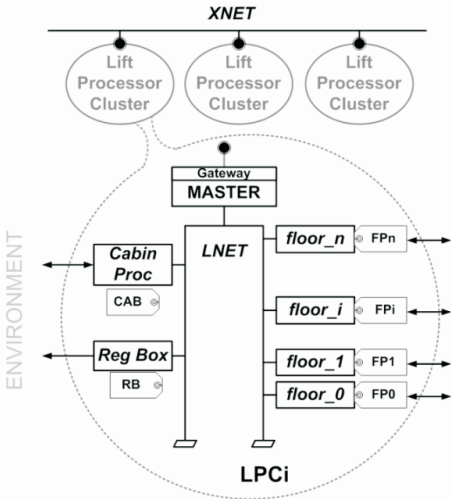


Fig. 2. Topology of the elevator (lift) system

Gateway for connecting to *XNET* bus is realized trough master node. 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, and to observe the behavior of the system under the injected faults.

This kind of flexibility is a consequence of topology the system i.e. his distributed pattern. This topology in the communication interface gives us possibility to insert the fault injection tool at any point in *LNET* or *XNET* bus. In this way, almost all possible hardware faults can be easily simulated. Hence, in this way we can simulate most of hardware and software faults. Additionally, the tool can be used in development phase as simulation tool.

### 2.3 Possible scenario

Manipulation with tool include the following steps:

1. The execution starts at the microcontroller subsystem.
2. Running the program on the PC activates the process of initiating communication with the target USB device, or a fault injector

3. After successfully establishing a connection to a USB device opens a window that is given in Fig 4. The appearance of the window is correlated with the frame structure shown in Fig 3, which is reflected in the following is presented as *CAB-STATION1*, *RB* as *STATION2* and so on. All fields are one byte in size and graphics are displayed at the bit to allow for manipulation of content on the level of bits. This setting allows the status of sensors, actuators and other parameters according to the principle: 0 - it OK and 1 - it NOT OK.

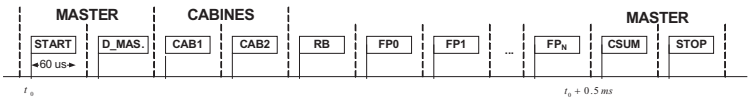


Fig. 3. Frame structure by LNET bus

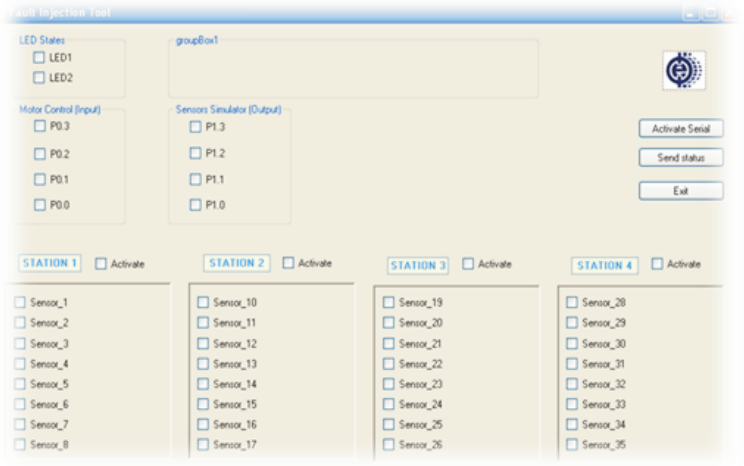


Fig. 4. Screenshot GUI

After completion of the message content in accordance with the desired test vector, the user, if the byte participate in such byte communication, the corresponding check box next to name *STATION X-Activate* is marked. Otherwise, instead of the defined message content tells the controller that in the given time slot a byte would be not sent. After preparation of the complete test sequence, pressing the Send button, status of the entire contents of 16 bytes are forwarded via it USB interface to the microcontroller. The microcontroller, in accordance with the described *TDMA* sequence forwarded received bytes via *RS485* connection to the lift-processor

The monitoring SUT test can be achieved in two steps:

1. First, activate the window for monitoring elevator processor in the same GUI, (direction 2 on the Fig 1.), or

2. Second, activating a software on another PC or laptop for collection data and stores it in the monitored data manager (direction 1 on the Fig 1).

After all the faults have been injected and gathered all the data from the target system, the controller finishes the experiment, deactivating the monitor. The graphical user interface is written in C #. Screenshot of this program is given on the Fig. 4.

### 2.4 Place and Role of the Toll in the Lift Systems

Quite apart from the effects of topology flexibility of the elevator, the second flexibility is reflected in a number of communication peripherals embedded in the C8051F340 microcontroller [5], such as *USB, UART1* and *UART2, SPI, SMBus*, 40 Port I/O. It gives us ability to provide various interfaces to system tools, and therefore insertion and monitoring of the system failures at different levels of the mod-el system shown in Fig. 2.

The first level is to have active Master for a lift-processor, and the rest of the system being simulated by the tool, shown in Fig. 5 a). In this case the rest of the system refers to all nodes in the *LNET* bus (cab, register box, floor nodes) and electric motor drivers. Tracking simulation results is through the interface between the lift-processor and *PC*. After confirming the validity of *Master*, and the correct status of the rest of the system, the next step is the insertion faults. The aim of this step is to track whether the lift processor has inserted an adequate response to the faults in terms of activation code for recovery or the introduction of the system in a safe condition by turning off the engine and brakes. A set of faults that can be injected at this stage are in accordance with lists of events which are of interest to the system, poor synchronization, the error resulting from the encoding techniques, or just some of the nodes is not alive.

The second level is the installation of individual nodes to *LNET* bus and reducing the role of tools in the simulation for some, not built, nodes, one or more, shown in Fig. 5 b). And, the third level is the simulation of the electric motor drivers a lift (three phase asynchronous motor).

### 2.5 Software on the microcontroller

Software tool is implemented at two levels:

**I) Kernel level** - dedicated kernel has designed the sys-tem time (tick), i.e. time interrupt the system clock 250*us* was implemented using Timer2 and the internal oscillator nominal frequency of 12 MHz. This time interval is a basic system for planning the execution of tasks and functions (Fig. 6) of the timeout.

USB module was developed as an upgrade *Silicon Laboratory USBXpress* ®[6] Development System, which provides a complete software solution for the C8051F340 device and the host PC to establish communication through*USB*.

By introducing a special external memory in the form of *SD* (*Security Data*) card, full potential of devices such as self-monitoring system and keeping records (dump file) for

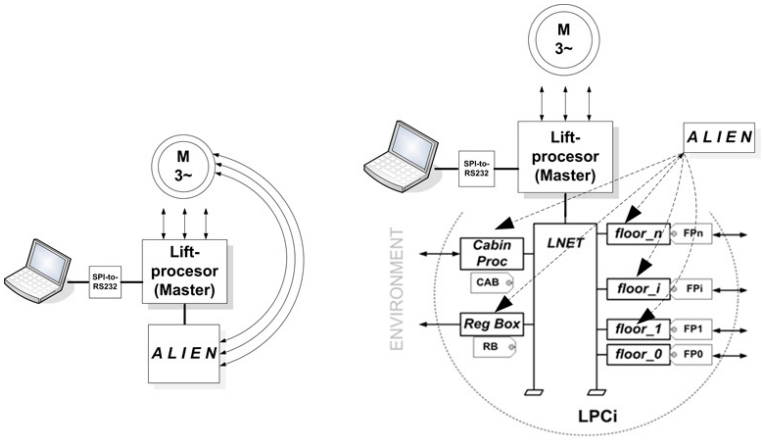


Fig. 5. Simulation versions

all events in the system, is achieved. *SD* Interface Firmware is low level interface that allows the rest of the system to access the *SD* memory cards. *SD* using *SPI* interface for communication in the 4-wire mode.

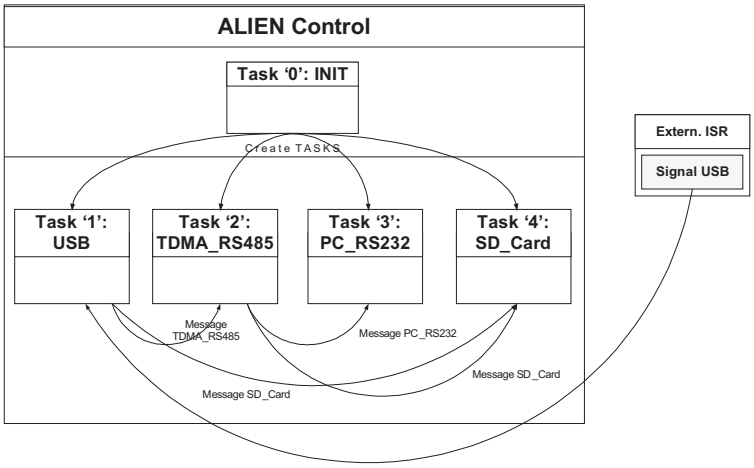


Fig. 6. Declaration of the task and the statechart diagram

**II) Tasks level** - software for the microcontroller is written in assembler and C51 and contains the following software tasks [7]:

1. Initialization task to the microcontroller and the system
2. USB task which allows communication between the PC and the fault injector
3. TDMA_RS485 task that generates a framework and distribution of bytes in the frames and communicate with SUT via RS485 interface

4. PC_RS232 task for communication with PC.

5. SD_Card task to store status bits and bytes of communication to SD card

At the end, tool is a hardware and software completed and tested. Test sequence is written in ASCII code to help identify the content of information.

## 3 Conclusion

This paper gives an overview of the tool ALIEN and de-scribed its place and role in the creation of a reliable lifts systems. A number of advantages are achieved. First, analysis of response to the inserted system faults in real work-ing conditions. Also, critical testing without equipment damage or endangering life. As a result of implementation tool we are able to detect the fault, identify the system component affected by the fault, and take an appropriate recovery action which may involve system reconfiguration. Also, using this tool, we came to the important knowledge about potential problems in the global system, and ways of their solving, which has provided us the opportunity to make an optimal compromise between performance, reliability, and cost.

## References

[1] L. L. PULLUM, *Software Fault Tolerance Techniques and Implementa-tion*, Norwood, MA, Artech House, Inc.2001

[2] http://strazzere.blogspot.com/2010/04/glossary-of-testing-terms.html

[3] J. ARLAT, M. AGUERA, L. AMAT, Y. CROUZET, J.-C. FABRE, J.-C. LAPRIE, E. MARTINS, D. POWELL, *Fault Injection for Dependability Validation: A Methodology and Some Ap-plications*, IEEE Transactions On Software Engineering, Vol. 16, 2 (1990), 166-182.

[4] D. TANG AND R.K. IYER, *Experimental Analysis of Computer System Dependability*, in: Fault-Tolerant Computer System Design, (D.K. Pradhan, ed.), Prentice-Hall Prof. Tech. Ref., Upper Saddle River, N.J., pp. 282-392.

[5] https://www.silabs.com/Support%20Documents/TechnicalDocs/C8051F34x.pdf

[6] *USBXpress*®, www.silabs.com

[7] B. PETROVIĆ, G. NIKOLIĆ, *Software development and testing tool for distributed elevator system*, Proceedings of X Triennial International SAUM Conference on Sys-tems, Automatic Control and Measurements SAUM 2010, Ni, Ser-bia, November 10-12, 2010, 296-299.