FACTA UNIVERSITATIS
Series: **Automatic Control and Robotics** Vol. 8, N° 1, 2009, pp. 45 - 65

DESIGN FOR TESTABILITY OF REAL-TIME SYSTEMS FOR INDUSTRIAL PROCESS CONTROL

UDC 621.11 681.5

Milun Jevtić, Milunka Damnjanović

Department of Electronics, University of Niš, Faculty of Electronic Engineering,
Aleksandra Medvedeva 14, 18000 Niš, Serbia
E-mail: (milun.jevtic; milunka.damnjanovic)@elfak.ni.ac.rs

Abstract. *Real-time-systems testing for the reliable functioning and protection of the system environment from damages is considered in this paper. A systematic procedure for real-time systems design with emphasis to system testability implementation is considered. The approach to design for testability (DFT) of hard real-time systems by monitoring is described. Different applied techniques for on-line testing on circuit-, system- and application-level are discussed. The modifications of some techniques made in order to accomplish the effective trade-off between space/time overhead and the cost of dependable real-time-system, are considered too. This paper also describes a realization run-time monitoring of real-time systems that can be used to verify formally some properties in design time, and to enable run-time checks. The goal of real time monitoring is to keep system performance within a range that does not change the order and timing of events.*

Key words: *Design, Real-time systems, Testability, Run-time monitoring, Fault tolerance, Time redundancy*

1. INTRODUCTION

Increased emphasis on shortening the time to first customer delivery, improving quality, effectively managing design complexity, and reducing overall life cycle costs drives designers to find more efficient ways to perform design verification and prototype test. The traditional design methodology of digital circuits and systems assume the test procedure and test sequence generation after the system or circuit logic design is completed. However the complexity of recent circuits and systems caused the need for too long time for test sequence generation or made it impossible. From another point of view, system testability affects the cost of prototyping and production testing and is very closely related to system reliability. During hardware-software integration, unanticipated iterations arise

Received May 14, 2010

that can cause significant schedule delays. Detecting and isolating these events or problems can add weeks or months to tight schedules [1], [2]. To solve this problem, circuit designers use design-for-testability (DFT) methodology [3], [4], [24]. A designer can directly affect a system's degree of testability and diagnosability by considering its test and diagnosis requirements as design requirements, not as test requirements decoupled from the design process.

To be successful, DFT must deal with test problems in all phases of the product's life [5]. Designers and test engineers can introduce test requirements and strategies early in the design cycle. They can also use the procedures that some methodology provides for test strategy selection and DFT implementation at the chip, multichip module, board, and system levels.

DFT strategy selection and implementation activities operate on successive refinements of a design, starting at the system specification level and continuing to the hardware and software physical-implementation level.

Testability is of special importance in RTS (Real-Time Systems) - systems whose correctness depends not only on their logical and functional behaviour, but also on the timing properties of this behaviour. They can be classified as hard RTS, in which the consequences of missing a deadline may be catastrophic, and soft RTS, in which the consequences are relatively milder.

Hard RTS testing include testing of functional behaviour and testing of timing properties. It is necessary to obtain it as on-line testing - system testing during its functioning and without degradation of system properties.

The application of the Real-Time-Systems (RTS) to control and monitor the processes like industrial, chemical or other processes dangerous for humanity and human environment and/or processes which can cause very expensive damages, states the additional specific requirements to designers [6]. Namely, it is important not only to accomplish properly monitor and control functioning with high reliability, but to predict the system behaviour in case of fault and error occurrence. For that reason, during the designing of these RTS, in addition to classical reliability consideration, the stabile provision of properly functioning hardware and system software resources (*dependability*) and the ability of a system to continue to function, perhaps at reduced capacity, in the presence of faults (*responsiveness*) must be considered.

One of the goals during real-time system designing process is to create a predictable real-time system. So, RTS testing is very important for the properly functioning verifications for the fabricated circuit or system, and for that performances achieving as well [7], [8]. The performance achieving states the necessity of testing (test procedure driving) during the circuit or system functioning in real time. This is known as *on-line testing*, in contrast to *off-line testing* when the circuit or system under test is not in working state but in special test environment.

Most test techniques work effectively by reducing the complexity of the circuit to be tested or by increasing controllability and observability.

To make the on-line testing possible, a circuit or system must have self-test possibility. So, the Built In Self Test (BIST) [9] was implemented on both - circuit and system level. RTS testing includes hardware, software and real time characteristics checking (real-time characteristics satisfaction).

Based on on-line testing, it is easy to accomplish the system function properly after some faults are detected in the system because permanent faults cause only a small fraction of all detected errors, as compared with transient faults [10]. Transient faults are induced mainly by outside disturbances such as power jitter, electromagnetic interference, ionization due to cosmic rays, or alpha particles from packaging materials. Of course, there is a class of intermittent faults - faults that occur from time to time, but in RTS they are classified either as transient faults or as permanent faults (in case of multiple fault occurrence in defined time-interval).

Here we identify critical design points and outline some practical solutions that refer to efficient on-line detectors (detecting errors during system operation) and error-handling procedures. The on-line testing techniques are to be considered as very important in finding the trade-offs between real-time system characteristics degradation caused by test accomplishing, and system cost. Some new modifications of the existing techniques for these purposes are presented in this paper. Also, a systematic procedure for design of RTS with emphasis to system testability realization is considered. An approach to DFT of hard RTS by monitoring is described and some base functions for its implementation are defined.

2. Conventional Strategies for Testing Real-Time Systems

Before we start considering on-line testing techniques, let's consider base strategy of testing RTS. [23]

There are at least three primary strategies for testing and evaluating control systems [8].

The sequential test environment shown in Figure 1. is the least complex. This strategy uses stored scenario data to drive the system under test. Test results are stored and analyzed.

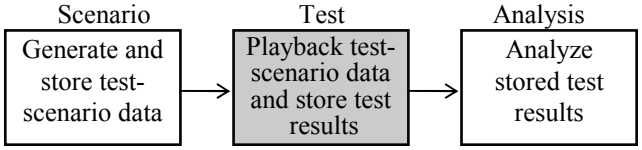

Fig. 1. Sequential test environment (Real-time task)

In this case, the only process running in real time is the system under test. The stored scenario data is generated off line before test execution. The stored scenario data can be either recorded operational and environmental data, or it can be simulated data produced by a scenario generation process.

This approach has some disadvantages:

- The success or failure of the test is unknown until the analyses are completed. Analyses are completed some time after the test is completed. Depending on the schedules at the test facility, it may be necessary to tear down the test setup and subsequently rebuild it if more tests are required.
- The scenario data must be rebuilt if a test needs to be suspended before completion because problems are observed. Generating a scenario can be a lengthy

process. If this is not done in real time, there could be significant down time while a new test scenario is generated.

- Test scenarios cannot be changed dynamically. It is not possible to observe the test in progress and ask "what if" questions.

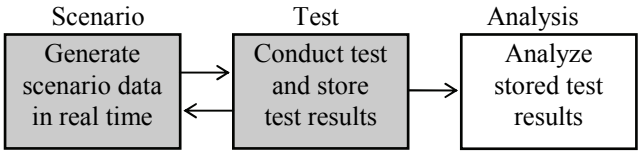

Fig. 2. Real-time scenario generation test environment
(Real-time tasks)

A more complicated simulation test environment involves conducting the control system test in real time as the scenario data is generated. This simulation test environment is called the real-time scenario generation test environment and is illustrated in Figure 2. In such a simulation environment, the data analysis is done off line, but the scenario generation and actual test are conduced in real time.

The real-time scenario generation and analysis test environment in Figure 3. is the only simulation test environment where all three procedures (generate scenario data, conduct test and analyze test data) are executed in real time.

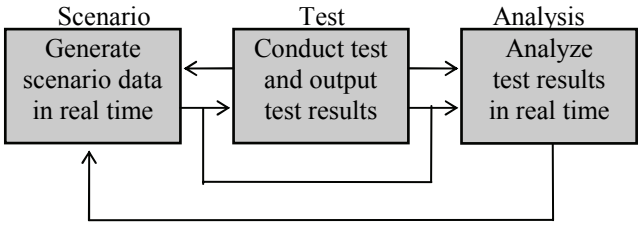

Fig. 3. Real-time scenario generation and analysis test environment (Real-time tasks)

It is good to concentrate on the real-time scenario generation and analysis test environment because this type of environment is useful in all phases of the life cycle.

For many large systems, all three testing environments are needed during the development life cycle. As a rule, the test approach to be implemented should maximize the amount of real-time testing.

The previous considerations are generally accepted when the systems with small number of processors are of interest. When the systems with great number of processors are the matter (like hyper-cub topologies), sequential testing consideration enters the scope of probabilistic model for the errors and test results [11].

3. RUNTIME MONITORING IN RTS

In software engineering, monitoring is the recording of specified event occurrences during program execution in order to gain runtime information that can not be obtained merely by studying the program text. The monitored information includes the runtime behaviour of the monitored program and pertinent information from the operating systems. In past years, researchers have used monitoring methods to gather information through different forms of instrumentation techniques for program testing and debugging, dynamic task scheduling, performance analysis, and program optimization [12], [13], [26].

The goal of real time monitoring is to keep system performance within a range that does not change the order and timing of events. Monitoring system is itself a system with real time constraints. Monitoring can be achieved at various levels. Low-level monitoring fetches every signal transmitted on the buses. High-level monitoring detects only process-level events. Instrumentation code is inserted manually or automatically as in breakpoint techniques. For events recognition, instrumentation code is inserted at specified points where it can generate the information pertaining to the events of interest. The monitored events should be events that are indicative of system behaviour.

Monitoring can occur either synchronously with application execution or asynchronously to the execution. *Synchronous checking,* or assertion checking, requires that the user add assertions to the application code. Assertions are checks to determine if, for a particular section of software, the relevant parts of the system state (for example variable values or I/O signals) are within the bounds needed for that section to operate properly. Assertions are placed directly in the application and can only be checked when encountered during execution. If more frequent checking is needed, *asynchronous checking* must be used. Asynchronous checking is done in an external process that receives events from the application.

Here, real-time monitoring is considered as system testing and debugging tool [25]. Namely, if we add the checking capability to the functions for events monitoring, which is not a great effort after HRTS integration and testing during the design process, it is easy to accomplish the testing of the timing characteristics of the events during the system's functioning [27],[30] .

There are three broad approaches to monitoring: hardware, software and hybrid approach.

3.1. Software Monitoring Approach

There are two software monitoring techniques.

1. One technique embeds the monitoring code inside the target program. This technique is not transparent.

2. The other technique embeds the monitoring code inside system kernels or treats the monitoring code as a process separated from the target code's process. This method is transparent but less flexible.

Software monitoring, which requires event detection and event data collection, can be described in the following ways:

- The target program detects the events and collects the event data.
- The target program detects the events and the monitor collects the event data.
- The kernel detects the events and the monitor collects the event data.

All three implementations have their advantages and disadvantages [32]. Implementing both event detection and event collection inside the target program is the easiest and most straightforward way to monitor the system. Performing event detection and event collection at the kernel level has two advantages over that at the program level: one is transparency, and the other is the reduction of monitoring perturbation. Transparency is easily achieved because users do not need to modify their target programs for event detection and event collection. Perturbation is reduced because context switching between the target program and the kernel is avoided. In contrast, the execution of event detection at the program level requires system call. This results in extensive context switching because system calls are implemented with interrupts to the kernel. However, event though the kernel level approach reduces the perturbation, the extra monitoring overhead imposed on the kernel still slows down the kernels and increases the target program's execution time.

3.2. Hardware Monitoring Approach

In general, monitoring hardware is used to monitor the runtime behaviour of either hardware devices or software modules. The former is used in the performance measurement of hardware devices, such as measuring cache accesses, cache misses, memory access time, total CPU time, total execution time, I/O requests, I/O grants, and I/O busy time. Monitoring software modules are used in debugging and in performance analysis of such software characteristics as program bottlenecks, dead locks, and the degree of parallelism. It seems that these two uses of monitoring are quite different, but the results of hardware performance measurement can also help in software performance analysis and debugging. For example, bottlenecks may be seen to arise from frequent references to the same memory location and deadlocks from messages locked within the communication network, while the degree of parallelism may be determined by dividing the total CPU time by the total execution time.

3.3. Hybrid Monitoring Approach

Hybrid monitoring consists of software instrumentation and hardware detection - that is, a compromise between hardware and software monitoring. The target program being monitored is first instrumented manually or automatically to generate events, then a hardware monitor is used to detect the events and to collect the corresponding event data. The hardware monitor can be designed as a permanent part of the target system during the design phase, or it can be an individual device or coprocessor integrated into a target system during the testing and debugging phase.

When the hardware monitor is implemented as a part of the target system, the steps for monitoring are as follows:

1. The event class to be monitored is defined and the memory for each class is allocated.
2. The instrumentation code is inserted into the target program.
3. The target program is compiled with the instrumentation code, and a monitoring symbol table is constructed for the instrumentation code.
4. The object code of the instrumented target program is linked and loaded so that the object code responsible for monitoring will flag the occurrence of an event, and write the

event, including the event class and its relevant information, into the preallocated memory space reserved in step 1 for that event class.

5. A hardware monitoring device snoops and matches the bus signal to the address allocated to the event class. An event is detected if the monitoring hardware device detects the address bus signal matching one of the preallocated addresses of the event classes.

When the hardware monitoring is implemented as a coprocessor, the steps for monitoring are as follows:

1. The event class to be monitored is defined and the memory for each class is allocated.
2. The instrumentation code is inserted into the target program.
3. The target program is compiled with the instrumentation code, and a monitoring symbol table is constructed for the instrumentation code.
4. The object code of the instrumented target program is linked and loaded so that the object code responsible for monitoring is executed by a dedicated monitoring coprocessor.
5. The monitoring coprocessor executes special monitoring instructions such as *write*, which quickly constructs and stores the event trace into a high-speed RAM inside the coprocessor.

3.4. Monitoring Implementation Functions

Both, hardware and software realized monitoring functions for embedding in the application code or in operating system kernel level are seen as a set of library procedures, usually in C language.

The observable events in a HRTS are specified by annotating real time programs with those events that are to be monitored at run-time. Examples of these events include start or completion of a program segment, and assignment to a state variable. A system constraint can be viewed as an assertion on the relationship between the occurrences of these observable events. The proposed approach distinguishes between a system constraint that is embedded in a RT program and a constraint that is monitored asynchronously by a separate task. Prototype implementation allows the specification and monitoring of both types of system constraints. In particular, the implementation supports annotating C programs with events and specifying system constraints.

In order to monitor system functioning through time by using the observable points, function *events* is needed. The initiation and completion of a sequence of program statements can be denoted by inserting this function to the source code as a label.

```
...
E_i(e_m, t_min, t_max)  ->
statement_S
E_k(e_j, t_min, t_max)  <-
...
```

Fig. 4 Code fragment with label events

The right pointing-arrow specifies the event e_i which denotes the start of the statement that follows (`statement_S`), and the left-pointing-arrow specifies the end of the previous statement. If $t_{min} = t_{max} = 0$, the function only stores the time of event occurrence, e.g. e_i related to the last occurrence of event e_m. For the defined t_{min} , $t_{max} \neq 0$, the function

provides checking whether or not the event, e.g. e_i, occurred inside the specified time interval, to initiate the adequate system functioning by the operational system.

This approach has two purposes: it separates the timing concerns from the functional specification of the program and it allows the expression of deadline and delay properties. It is assumed that (1) two occurrences of the same event can not happen simultaneously, (2) event names are unique across the system of tasks, and (3) there exists a single monotonically increasing clock, accessible to all tasks.

Our approach considers the use of monitoring information to aid in scheduling task in real-time environment [29]. The monitored information is fed back to the operating system for achieving an adaptive behaviour.

Particular types of the observable events are assignments of values to a variable referred to as watchable variable. If variable v is declare by

$$\texttt{watchable\_v v}(v_{min}, v_{max}, \Delta v)$$

any assignment of the value to it is considered as event, and if $v_{min}, v_{max} \neq 0$ and $\Delta v \neq 0$ is defined, checking is done if the value is inside the defined interval, and if the absolute change of the value related to the last previous value greater than Δv.

Event history is generated in the system by storing the times and/or values of watchable variables. For event history survey during the prototype behaviour testing and for event reconstruction during the system functioning, two other functions (on the software kernel level) are needed: `@(e,i)` and `@val(v,i)`. `@(e,i)` is the occurrence function which returns the time of `i`th occurrence of the event `e` (`i` can be negative for event referring backward). `@val(v,i)` is the function which returns the value of watchable variable `v` on `i`th event of its change.

3.5. RT Task Monitoring Scenarios

RT tasks can be divided into pre-emptive and non pre-emptive tasks. Concerning its execution time, pre-emptive tasks, unlike the non pre-emptive, do not have strict limits. Also, their possible failure in execution would not affect significantly the proper functioning of RTS. Therefore, the scheduler can pause the execution of such tasks when receiving execution request from some higher priority RT task. After the execution of high priority task scheduler continues the execution of previously paused task. On the other hand, non pre-emptive tasks execution failure, or execution outside given time limits can lead to whole RTS failure. Because of this, high priorities are assigned to these tasks, and they can not be paused while running.

Non pre-emptive RT tasks: Possible course of non pre-emptive task (T_i) execution is shown on Figure 5.

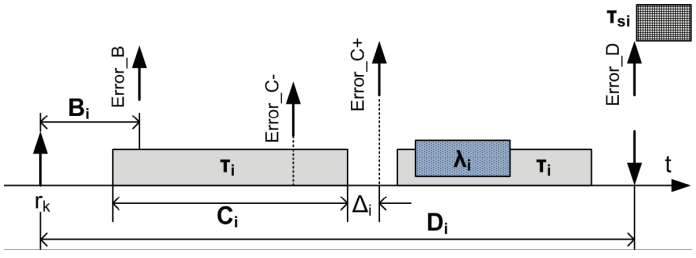

Fig. 5. Monitoring scenario of non pre-emptive task execution

From the moment – event r_k when the request for task τ_i execution occurred, allowed delay to starting the task execution can be checked at first. This is important for the tasks that do not initiate with some external interrupt event. These tasks are "set" in the queue for execution by some internal event. In the case of exceeding the interval B_i, monitoring timer-counter generates a hardware interrupt request, and error Error_B is detected. Another monitored time interval is task execution time (CPU time). For task execution time which is shorter than C_i (minimum required time for correct task execution), marker Error_C- is set. In the case of exceeding the task execution time $C_i + \Delta_i$ (maximum time for correct task execution) monitoring module generates interrupt request to detect error Error_C+.

Such monitor performs over each RT task. Upon detection of any of these errors, it is the policy of the planner and available time what will be taken. Restarting the same task or starting an alternative task (λ_i) execution which will overcome given situation can be done. For each task, deadline D_i for its execution should also be monitored. Special counter-timer is the most suitable for this purpose. In the case of its exceeding, interrupt request is generated and hardware-software security task (T_{si}) is started. This security task should recover RTS or place it in a safe condition.

Pre-emptive RT tasks: Monitoring of pre-emptive tasks τ_i (Figure 6) differs from the previous monitoring scenario. While its execution is stopped because of a higher priority task τj, its monitoring timer-counter should be stopped (during C_j).

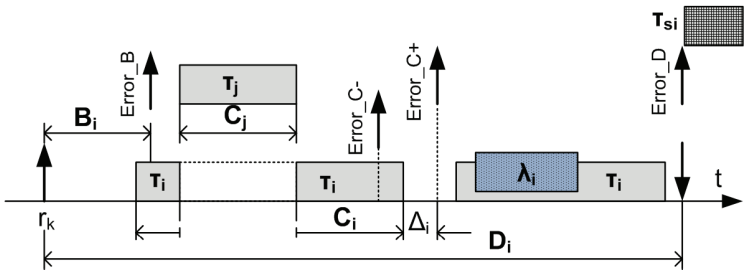

Fig. 6. Monitoring scenario of pre-emptive task execution

3.6. Hybrid On-line Process Monitoring Module

Depending on the application and environment, timing constraints imposed on a RTS vary widely. Here presented FPGA based monitoring module [33] would be applicable to each RTS determined to meet strict timing constraints imposed by the real world processes. FPGAs are chosen because of their low cost and ability for reconfiguration.

Posing the demand that *on-line* monitoring do not require significant CPU time and clumsy additional specialized hardware, this paper presents one way of FPGA implementation of hybrid on-line RTS monitoring. It is intended for RTS based on an industrial PC and Linux operating system which is widely accepted and available open source system in RTS.

The system is based on additional hardware module with 32 programmable timer-counters and interrupt logic. Each monitored process has assigned its own timer-counter. Timers-counters are used as devices for defining the moments of events' time occurrence as well as watchdog i.e. monitoring timers for checking the correct timing execution of the processes. In addition, simple software primitives for on-line monitoring implemen-

tation are realized. They can be activated from the desired place in application program code. For monitoring of the processes and tasks in RTS without modification of application program code, simple modification of the operating system task scheduler and dispatcher is predicted. Modification ensures that scheduler or dispatcher, with every change of process/task status, activates appropriate software primitive for controlling timer and checking the time constraints.

For minimal intrusion and using of CPU time during monitoring, hardware module for PCs PCI slot is realized as shown in Figure 7.

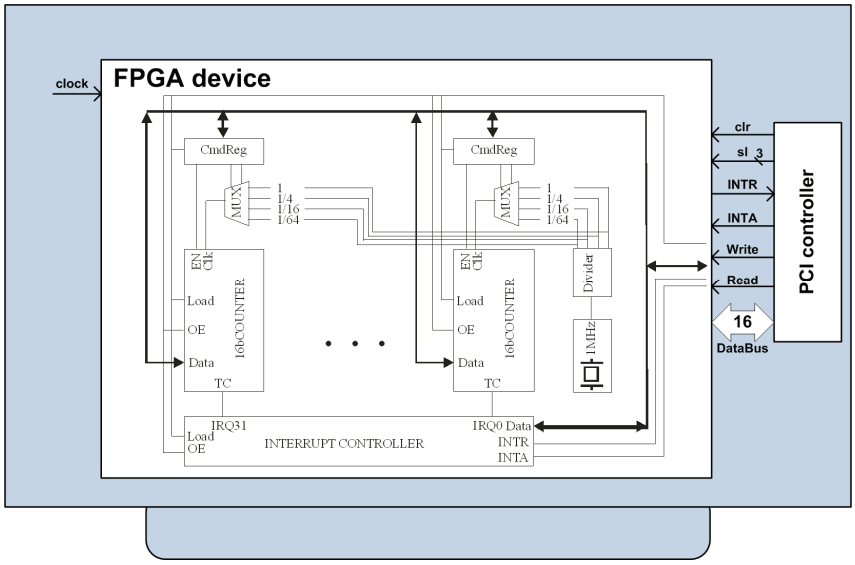

Fig. 7. PCI card with hybrid on-line monitoring module

From Figure 7 it can be seen that the interface from monitoring module to RTS consists of the following signals: DataBus, Read, Write, INTR, INTA, sl and clr. DataBus is a 16 bit bidirectional bus. It transmits the data from RTS to monitoring module and vice versa. RTS activates Read (Write) signals every time it needs to read data from (write data to) monitoring module. Monitoring module sets INTR (Interrupt request) signal every time any of currently executing tasks do not execute properly or execute outside of the required time interval. As a response, RTS reads the message from DataBus and sets INTA (Interrupt Acknowledge) signal. The message contains information about the interrupt nature and the ID of the task that caused interrupt. It is now scheduler policy to determine the actions that will be taken. When receiving Interrupt Acknowledge, monitoring module resets INTR signal and continues to monitor RTS.

Signal sl is 3 bit select used by RTS when selecting the register from which to read data (or selecting the register to write data to). The use of this signal will be explained later in more detail. Signal clr has a function of clear signal and it is used by RTS to reset monitoring module.

Monitoring module is controlled by software primitives from RTS and has the following functions:

- Setting the working mode of the timers-counters,
- Setting the time constraints,
- Enabling the timers-counters,
- Disabling the timers-counters,
- Reading the timers-counters,
- Timers-counters interrupt processing and
- Comparison of the timers-counters state with time constraints.

During the system verification phase monitoring system provides information about system timing characteristics and creates a log file. During the system operation it should detect deviations from predicted timing behaviour. These deviations could be the possible consequence of a failure in RTS. Thereby, monitoring system has two working modes. First mode refers to the system analysis. It performs with the purpose to measure the execution time of every RT task. The obtained information can be used for the future control of the RTS. In the second mode monitoring module has the function of built-in selftesting based on a watchdog function. It checks the upper and lower time limit at the tasks and periodic and quasi-periodic events level. The activation of each task initiates the procedure of starting its assigned timer-counter. Monitoring timer-counter sets to previously defined maximum task execution time and starts its countdown. If the excess of time interval occurs, monitoring module sets interrupt request. If the task is complete before time excess, timer-counter stops its countdown with the end of task execution. Monitoring module reads its state and checks whether the task is executed before the minimum needed execution time. If the task is executed in regular time intervals RTS continues to work. Otherwise, scheduler starts provided procedure for system recovery from detected error. In this way, predicted behaviour of HRTS is ensured.

Implemented system monitors up to 32 processes i.e. RT tasks and events that execute in parallel. The number of monitored processes is relatively small, but it should be said that HRTS in industrial applications do not have a lot of processes. But since our monitoring module for 32 processes requires only 23% of FPGA ALTERA EP2C35F672C6N resources, as will be seen later, the number of monitored processes can be easily expanded up to 150.

4. SYSTEMATIC APPROACH TO RTS DFT

If hardware/software co-design is assumed as a method for microcomputer systems design [5], [14], [28] the activities for testability achieving are as is shown in Fig. 8. As can be seen, the testability design activities are distributed through all design phases. During the system specification defining, the uniform testing requirements have to be stated, taking into account the requirements for system testing during

- system design,
- system production,
- system functioning and
- system maintenance.

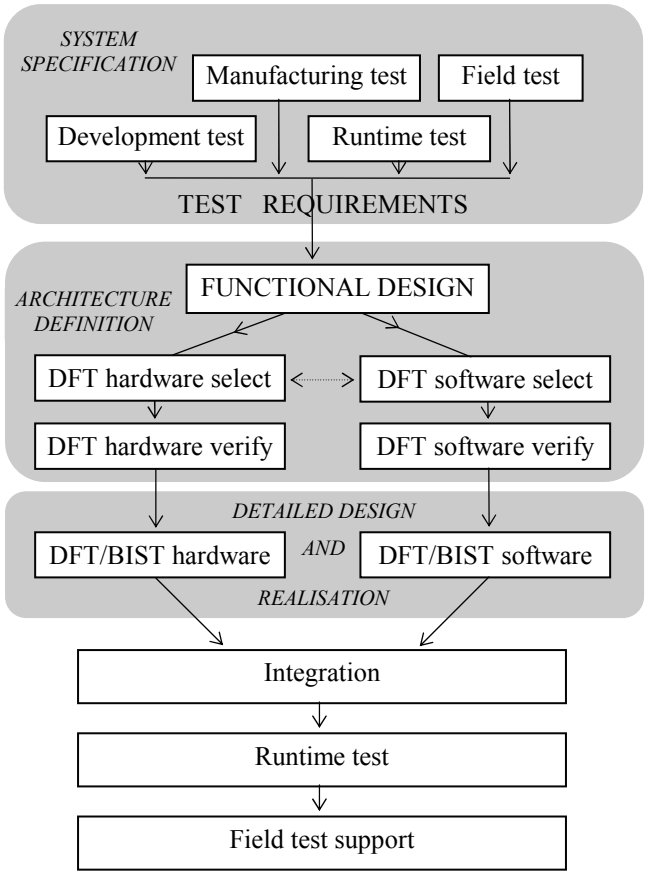

Fig. 8. Systematic Approach to Real-time System DFT

For efficient achieving of today's hard RTS testing requirements, it is necessary to use BIST approach on all levels - application, system, subsystem, board, and chip level. So, if principles of operation are considered, error detection mechanisms can be divided into corresponding levels (application, system and circuit level) [34].

Various on-line error detectors can be incorporated in the computer system [15]. The basic principle of these detectors is the use of redundancy in

- devices (hardware replication),
- information (redundant codes),
- software or
- execution time.

The functions from function-set for BIST implementation, defined on system level, are further developing into more detailed functions for hardware and software testing. Always looking for the tradeoffs between system redundancy and test efficiency, the BIST functions are implemented as hardware, software, or (in most cases) hardware-software

combination. The third case is dominant because it introduces minimal hardware/software redundancy for enabling testability with flexible test running. Even in BIST or scan technique implementation in hardware module or chip testing, to accomplish on-line (run-time) testing, what is needed is software control (start and break) for test running and, if wanted, test results checking.

In safety-critical RTS, the software testing phase may cost from three to five times as much as any other phase of the software life cycle. Usually, testing begins early in the development process, as test planning and specification overlap, to certain degree, with requirements specification.

Aspects of general testing strategies and techniques also apply to real-time software. However, additional requirements and difficulties characterize an effective testing process for a real-time software for several reasons:

- The software contains several modules and decision statements.
- Many modules use the same resources simultaneously.
- The same sequence of test cases may produce different outputs, depending on when the test is performed.
- System errors may be time dependent, only arising when the system and controlled environment are in a particular state that may be impossible to reproduce.
- Finally, reliability, schedule, and performance requirements are usually more critical than those for non-real-time software.

The strategy followed for testing an RTS is systematic and includes individual function, module, and bottom-up integration, using black-box and white-box disciplines.

5. REDUNDANCY IN RTS

Besides the basic function of Control RTS, different features such as the following might be required:

- high reliability (high probability of continuous proper function over a given, long interval),
- high availability (relatively low percentage of repair times),
- minimum time to recover from a detected fault,
- extremely low failure rates for short time periods,
- extremely high probability of transition to a safe final state after occurrence of a malfunction,
- easy and timely diagnosis of faults or defects (on-line diagnosis).
- safety and security (are the abilities to protect the system from inadvertent misuse or from malicious attempts to destroy system functionality,
- high dependability (the stable provision of properly functioning hardware and system software resources),
- responsiveness (the ability of a system to continue to function, perhaps at reduced capacity, in the presence of faults, also without delaying critical process deadlines).

These performances achievement is closely connected to redundancy implementation. Redundancy in space and time [31] as fundamental resources of the RTS (Figure 9.),

Otherwise, space-time trade-off considering has the cost-effective RTS as an aim, although time often plays the most critical role.

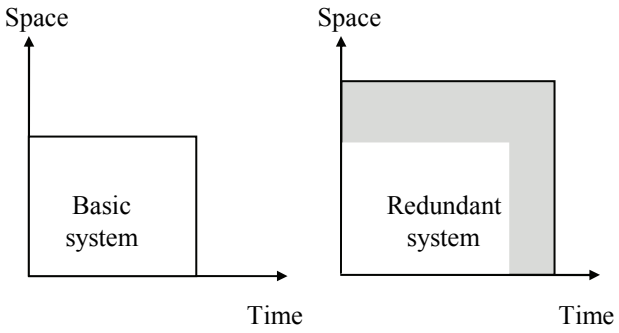

Fig. 9. Basic and redundant RTS

A global review of different redundant systems is shown in Fig. 10. Some approaches use great hardware (space) redundancy but need very low time redundancy to provide reliable functioning through error detection (error-correcting codes, duplex system, triple-modular redundancy, etc.). Others need great time redundancy (multiprocessor rollback and recovery, N-version programming, etc.)

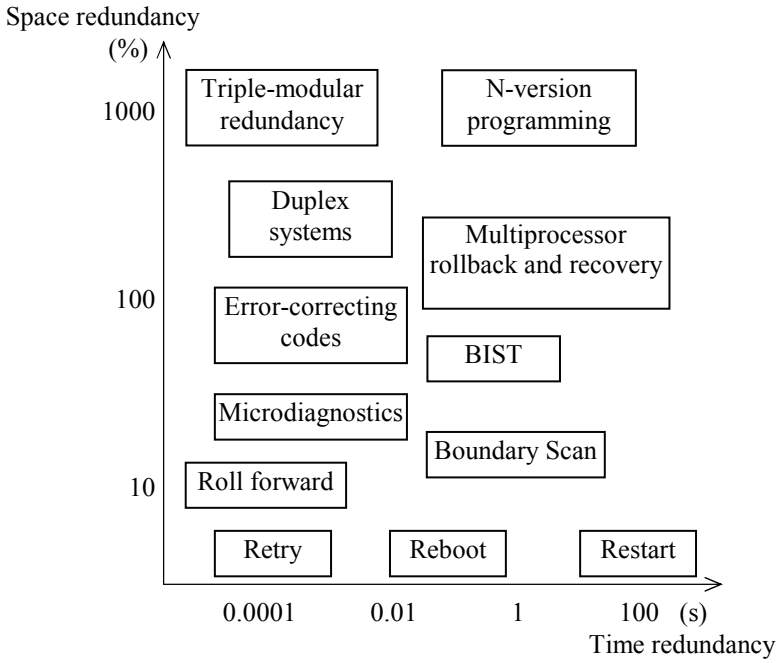

Fig. 10. Space-time overhead for redundant approaches

Error correcting codes, depending on code distance, detect and correct any number of bit errors.

Duplex system starts the error handling procedure in case of two different results occurrence.

If three identical components are used, i.e. in the case of triple modular redundancy, the results are compared via majority voting circuits. If one of the results differs from the other two, it is masked out.

N-version programming provides comparison of results obtained in N various versions of the program.

In backward recovery, an attempt is made to restore the system to a safe previous state.

For forward recovery, an attempt is made to correct the system state.

Boundary scan provides primary an easy testing on circuit and module (PCB) level in off-line mode. However, the on-line testing of particular modules can be accomplished in time intervals when they are not active. It needs small additional hardware and a diagnostic test-routine.

BIST, based on scan technique, is inadequate for on-line testing the modules that are continuously functioning but can easily be used in system dependability providing by minimal software expanding and insignificant processor-time consumption. On system level, BIST can be implemented in concurrent or nonconcurrent form.

In control RTS, BIST as general approach realized by different (even ad-hoc) techniques of self-testing is a very good solution.

6. On-Line Error Detectors – Error Detection Mechanisms

Various on-line error detectors can be incorporated in a computer system. The basic principle of these detectors is the use of redundancy in

- devices (hardware replication),
- information (redundant codes),
- software or
- execution time.

Taking into account principles of operation, we can divide error detection mechanisms into three levels:

- circuit,
- system (functional), and
- application.

6.1. Circuit level

Circuit-level detection mechanisms are implemented using error detection codes, self-checking circuits, and duplicated complementary circuits [16]-[19].

A circuit is self-checking for a set of faults **F** if for any fault in **F** there exists a valid input code that detects this fault. Scan and BIST techniques implemented in VLSI circuits support this detection mechanism [9], [20]. BIST technique implementation can result in

a significant space overhead, but it doesn't need slightly processor time for testing, except for the starting and essential test output analyzing

Error detection code applied to any functional unit provides error detection and correction. In general, a code can correct up to **r** bit errors and detect up to **e** additional bit errors if and only if

$$\mathbf{2r + e + 1 \leq H_d},$$

where $\mathbf{H_d}$ is the Hamming distance of the code (minimum number of positions in which any two code words differ). So, simple parity codes ($\mathbf{H_d} = 2$) assure detection of any single-bit errors and all multiple-bit errors with an odd number of bit position changes. But some other codes can detect and correct a number of errors: Hamming codes, Cyclic Redundant Codes (CRC), AN cod (special code for arithmetic units), etc. [21]

Circuit-level detectors are quite effective for some system blocks such as memories or communication subsystems. For other blocks the cost/error coverage ratio may not be satisfactory for many applications, and thus system-level detectors have recently received much attention.

6.2. System-level

System-level detection mechanisms can be regarded as "guardians" around the system operation that check out invalid activities or information at a higher level than the previous ones. They verify system operations by functional assertions on program control flow, memory accesses, and so on. Thus, these detectors verify the correctness of the system operation by checking its various general properties. The most effective system detectors use hardware replication or time redundancy.

Hardware replication of the whole system (or its main modules) allows us to compare various signals in replicated system modules.

Since hardware replication is expensive for many applications, an alternative solution is developed based on a comparison of results obtained in repeated computations (time redundancy). These computations can be performed for the same program or its various versions (**N**-version programming).

Many system-level error detection mechanisms that result in low hardware and time redundancy [22] are presented. They are based on monitoring various system characteristics from valid memory access checking to sophisticated control flow checking.

A control flow graph (CFG), with nodes representing a program unit (a block of instructions) and arcs specifying possible paths within the program, defines the structure of a program. Many control flow checking schemes are based on associating signatures (tags) with CFG nodes. These schemes use different program units as nodes, different definitions of signatures, and monitoring techniques.

Assigned signatures are associated arbitrarily with the program nodes. Derived signatures are calculated from the contents of nodes. Schemes that use assigned signatures check for node executions in an allowed sequence, while techniques based on derived signatures verify the contents of nodes and the succession of nodes. The verification process requires a description of the valid program control flow. This data can be included in the monitored system program or in the watch-dog program (a mixed solution is also possible).

We can implement assigned signature schemes as software or hardware watchdogs. In the first case the system invokes a checking subroutine each time signature verification or updating is needed. The hardware watchdog program, which is designed around the monitored system's CFG, can be based on an interpreted or a compiled tracing technique. Each time a tag is received for checking, the interpreting program refers to the CFG structure stored in the watchdog memory. Compiled graph tracing is faster. It also does not require a separate data structure to represent the CFG because it is embedded into the watchdog program. (Each node of the CFG is represented by a watchdog program segment.) External processors (single-chip computers) or some partially used resources of the monitored system can serve as the watchdog circuit.

Many systems are realized with derived-signature control flow checking capability. The simplest detectors of this class are watchdog timers (WDTs) that check program execution time [22]. WDTs restart sequences embedded into application program initialize WDTs periodically. If not initialized (due to an error) within a specified time, the WD Timers generate error signals. That signal usually activates the microprocessor non-masked interrupt causing the system restart.

Assume a WDT checks the lower (T-T_w) and upper (T) bounds of a program segment's (CS_i) execution time (t_i) and parameters T and T_w can be fixed for all segments. We can estimate the probability $p_{ij}(T_w)$ that an erroneous branch from segment CS_i into segment CS_j (with execution time t_j) is detected by the WDT:

$$p_{ij}(T_w) = [(t_i + t_j - T)^2 + (T - T_w)^2] / (2\, t_i\, t_j) .$$

The optimal situation arises for $t_i = t_j = T\text{-}T_w/2$. By decreasing T_w, we can achieve quite good error coverage. However this requires knowledge of program execution time (estimated with special programs). Most popular WDTs check only the upper bound ($T_w = T$). In this case maximal value of $p_{ij}(T_w=T)$ is 0.5. One such WDT is a part of monitoring module shown in Figure 7.

6.3. Application-level

Application-level detection mechanisms are related to the realized algorithms in the system and properties of generated results (acceptable ranges of variables, reasonableness of computation results). For many systems we can find effective checking rules at the application level (algorithm-based techniques), for example, by means of assertions. An assertion is an invariant relationship between program variables or system output signals. Assertions can be defined by making use of various properties of the problem or algorithm. They are usually based on the inverse of the problem, the range of variables, and the relations between them. Sometimes this solution involves the introduction of a check variable. When the system controls a physical object, the behavior of this object (in response to system stimuli) may be described by some rules that the detector can check.

Application-level detectors may generate false alarms (in the absence of errors) due to imprecise checking rules or round-off errors. So we must check for the preservation of these rules within tolerance.

7. TEST PROCEDURES IN RTS

Some on-line testing techniques require software support, so it is necessary to have the self-test task in an RTS. It means that scheduler must provide time for self-test. The global self-test task is realized through the individual self-test task of the system elements. The individual self-test task can have different priorities and different running periods regarded to its influence on reliable system functioning.

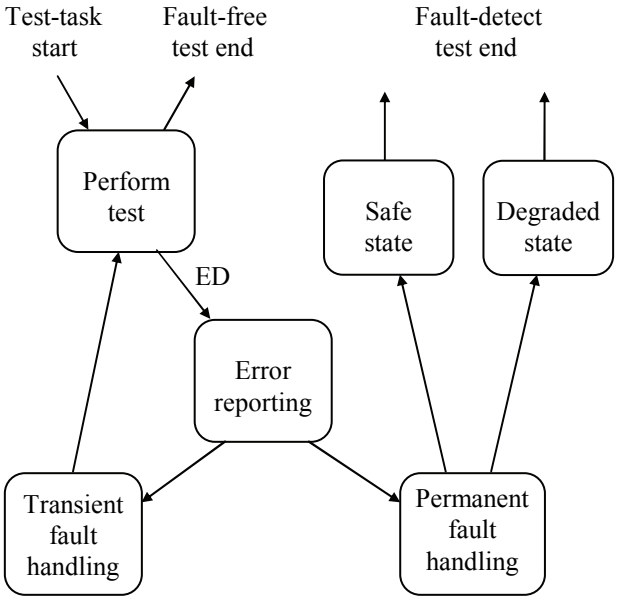

Fig. 11. State diagram of an test procedure

Any self-test procedure, disregarding the starting way, can be represented by state diagram shown in Fig. 11. During the error reporting phase, after error detection (ED), the fault causing that error is reported as transient fault. But, if the same error is reported **k**

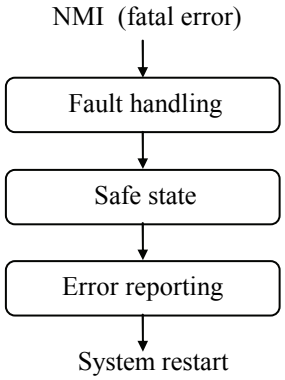

Fig. 12. State diagram of a fatal error procedure

times (practically, **k** = 2) within a specified time window, the fault will be labeled as permanent. In case of permanent fault detection, RTS must be set to safe state or degraded state, and the system has to start functioning in the way predicted for that situation.

Some faults in RTS can be fatal for the system and its environment. They are usually detected as hardware event on nonmasked processor interrupt input having the task supporting the system set to safe-security state (Fig. 12.). In fact, these faults require some kind of system restart.

8. CONCLUSION

Since design for testability has become a necessity in electronic system design, a systematic procedure for RTS design with emphasis to system testability implementation must be considered. Here, it is highlighted that to be successful, DFT must deal with test problems in all phases of a product's life. To enable this, designers and test engineers can introduce test requirements and strategies early in the design cycle.

The on-line testing techniques considered in this paper are important for realization of the RTS which must have predicted functioning in case of failure (safety hard real-time system). Such systems often need sensors and actuators with built-in self-test (like smart sensors).

Considered modifications of some test techniques have the task to enable easy, fast and cost-effective testing of the elements of RTS and the entire RTS, with insignificant processor-time consuming. Good combination of these techniques on circuit-, system-, and application-level, can result in RTS with desired characteristics and cost-effectiveness.

Of course, a convenient methodology for design process automation (which is not considered here) must be used with these testing techniques.

REFERENCES

1. Bernd Koenemann, Ben Bennetts, Najmi Jarwala, Benoit Nadeau-Dostie, *Built-In Self-Test: Assuring System Integrity,* IEEE Computer, Vol. 29, No. 11, November 1996, pp.39-45.
2. Manthos A. Tsoukarellas, Vasilis C. Georgianis, Kostis D. Economides, *Systematically Testing a Real-Time Operating System,* IEEE Micro, Vol. 15, No.5, October 1995, pp.50-60.
3. S. Bhawmik, P. Palchaudhuri, *DFT Expert: Designing Testable VLSI Circuits,* IEEE Design & Test of Computers, Vol. 6, No. 5, October 1989, pp. 8-19.
4. M. Abramovici, M. Breuer, A. D. Friedman, *Digital System Testing and Testable Design,* Computer Science Press, New York 1990.
5. Harold P.E. Vranken, Marc F. Witteman, and Ronald C. van Wuijtswinkel, *Design for Testability in Hardware-Software Systems,* IEEE Design & Test of Computers, Vol. 13, No.3, Fall 1996. pp.79-87.
6. Lemos, A. Saeed, and T. Anderson, *Analyzing Safety Requirements for Process-Control Systems,* IEEE Software, Vol. 12, No. 3, May 1995, pp. 42-53.
7. Scott Davidson, *Software Tools for Hardware Test,* IEEE Computer, Vol. 22, No. 4, April 1989, pp. 12-14.
8. K. D. Shere, R. A. Carlson, *A Methodology for Design, Test, and Evaluation of Real-Time Systems*, IEEE Computer, Vol. 27, No. 2, Febr. 1994, pp. 35-48.

9. Vishwani D. Agraval, Charles R. Kime, Kewal K. Saluja, *A Tutorial on Built-In Self-Test, Part 2: Applications,* IEEE Design & Test of Computers, Vol. 10, No. 2, pp. 69-77, June 1993.
10. Janusz Sosnowski, *Transient Fault Tolerance in Digital Systems,* IEEE Micro, Vol. 14, No. 1, February 1994, pp. 24-35.
11. Douglas M. Blough, and Andrzej Pelc, *Diagnosis and Repair in Multiprocessor Systems,* IEEE Transactions on Computers, Vol.42, No. 2, February 1993, pp. 205-217.
12. Beth A. Schroeder, *On-line Monitoring: A Tutorial*, IEEE Computer, Vil. 28, No. 6, June 1995, pp.72-78.
13. Bernhard Platner, *Real-Time Execution Monitoring,* IEEE Trans. Software Eng., Vol. SE-10, No. 6, Nov. 1984, pp. 756-764.
14. M. Jevtić, W. Weber, B. Đorđević, *A Database For Integrated Design of Microcomputer Based Systems*, Facta Universitatis, Series: Electronics and Energetics, Vol.4, No.1., pp. 13-27, 1991.
15. M. Jevtić, M. Damnjanović, *Testing of Digital Systems in Real-Time Applications,* Proceedings 20th International Conference on Microelectronics - MIEL'95, Vol.2, pp. 835-840, Ni{, Serbia, Yugoslavia, September 1995.
16. J. McClaskey, *Design Techniques for Testable Embedded Error Checkers,* IEEE Computer, Vol. 23, No. 7, July 1990, pp. 84-88.
17. Pat McHugh *IEEE P1149.5 Module Test and Maintenance Bus,* IEEE Design & Test of Computers, Vol. 9, No. 4, pp. 62-65, Dec. 1992
18. M. Jevtić, M. Damnjanović, G. Cvetković, *On/Off Input-Output Module with Built-In-Self-Test,* Proceeding of YU INFO '95, Brezovica, April 1995.
19. S. Kundu, S. M. Reddy, *Embedded Totaly Self-Checking Checkers: A Practical Design,* IEEE Design & Test of Computers, Vol. 7, No. 3, August 1990, pp. 5-12.
20. S. Narayanan, R. Gupta, *Optimal Configuring of Multiple Scan Chains,* in IEEE Transactions on Computers, Vol.42, No. 9, September 1993, pp. 1121-1131.
21. E. Fujiwara, D. K. Pradhan, *Error-Control Coding in Computers,* IEEE Computer, Vol.23, No. 7, July 1990, pp. 63-72.
22. A.Mahmood and E. J. McClaskey, *Concurrent Error Detection Using Watchdog Processors,* IEEE Trans. on Computers, Vol. C-37, No. 2, 1988, pp. 160-174.
23. M. Jevtić, M. Damnjanović, *TESTING OF DIGITAL SYSTEMS IN REAL-TIME APPLICATIONS*, Proceedings 20th International Conference on Microelectronics - MIEL'95, Vol.2, pp. 835-840, Niš, Serbia, September 1995
24. Milun Jevtić, Milunka Damnjanović, *AN APPROACH TO DESIGN FOR TESTABILITY IN HARD REAL-TIME SYSTEMS*, Proceedings 21th International Conference on Microelectronics - MIEL'97, Vol.2, pp. 849-852, Niš, Serbia, September 1997.
25. Milun Jevtić, Milunka Damnjanović and Vladimir Živković, *A SOLUTION TO RUN-TIME MONITORING OF REAL-TIME SYSTEMS*, Proceedings of First Conference on Electrical Engineering & Electronics, December 1998. Gabrovo, pp. 272-277.
26. Dennis K. Peters, David Lorge Parnas, *Requirements-Based Monitors for Real-Time Systems*, IEEE Transactions on Software Engineering, Vol. 26, No. 2, February 2002, pp. 146-158.
27. Milun Jevtić, Marko Cvetković, Sandra Brankov, TASK EXECUTION IN REAL-TIME SYSTEMS FOR INDUSTRIAL CONTROL AND MONITORING, *Electronics*, Faculty of Electrical Engineering University of Banjaluka, vol. 6, no. 2 december 2002, pp. 56-61
28. G. Lima and A. Burns, "*An Optimal Fixed-Priority Assignment Algorithm for Supporting Fault-Tolerant Hard Real-Time Systems*", IEEE Transaction on Computers, Vol. 52, No. 10, pp. 1332-1346, October 2003.
29. Milun Jevtic, Volker Zerbe, Sandra Brankov, *"Multilevel Validation of On-Line Monitor for Hard Real Time Systems"*,Proc. 24th International Conference on Microelectronics - MIEL 2004, Vol. 2, pp. 755-758, Nis, Serbia and Montenegro, 16-19. may, 2004.
30. Tsai J., Fang K., Chen H.: "*A Noninvasive Architecture to Monitor Real-Time Distributed Systems*", IEEE Computer Society, Vol. 23, pp. 11-23, 2004.
31. S. Đošić and M. Jevtić, "*Scheduling in RTS Using Time Redundancy for System Recovery After Faults*", Proceedings of papers, Indel 2004, Banja Luka, pp. 146-149, November 2004.

32. Nelly Delgado, Ann Quiroz Gates, and Steve Roach, *A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools*, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 30, NO. 12, DECEMBER 2004, pp. 859 - 872
33. Jovanovic B., Jevtic M.: "*Module for run-time monitoring in PC hardware based real-time system*", Proc. Int. Scientific Conference - Unitech 09, Gabrovo, Bulgaria, 2009.
34. Christian Colombo, Gordon J. Pace and Gerardo Schneider, *Safe Runtime Verification of Real-Time Properties,* Book: Formal Modeling and Analysis of Timed Systems, Springer Berlin / Heidelberg, ISSN 0302-9743 (Print) 1611-3349 (Online), Volume 5813/2009, pp. 103-117.

PROJEKTOVANJE ZA TESTABILNOST SISTEMA ZA RAD U REALNOM VREMENU ZA UPRAVLJANJE INDUSTRIJSKIM PROCESOM

Milun Jevtić, Milunka Damnjanović

Ovaj rad razmatra proveru sistema za rad u realnom vremenu u cilju pouzdanog rada i zaštite sredine sistema od oštećenja Razmatra se sistematičan postupak za projektovanje sistema za rad u realnom vremenu sa akcentom na implementaciji mogućnosti provere sistema. Opisan je pristup projektovanja za testabilnost (DFT) rigidnih sistema za rad u realnom vremenu pomoću nadzora. U tekstu će biti reči o različitim primenjenim tehnikama on-line provere na nivou kola, sistema i primene. Modifikacije nekih tehnika u cilju postizanja boljeg odnosa između prostornih i vremenskih rešenja i troškova pouzdanog sistema za rad u realnom vremenu, takođe su razmatrane. Ovaj rad takođe opisuje realizaciju nadzora rada sistema za rad u realnom vremenu koji se može koristiti za formalnu verifikaciju nekih vremenskih karakteristika projekta, kao i da omogući provere rada. Cilj nadzora u realnom vremenu je da održi performanse sistema u opsegu koji ne menja raspored i vreme procesa.

Ključne reči: *Projektovanje, Sistemi za rad u realnom vremenu, Testabilnost, Nadzor rada, Tolerisanja otkaza, Vremenska redundansa*