Overview of Mechanisms for Improving Reliability of Embedded Real-Time Systems

Josef Strnadel, Brno University of Technology, Faculty of Information Technology
Božetěchova 2, 61266 Brno, Czech Republic
e-mail (strnadel@fit.vutbr.cz), phone (+420541141211), fax (+420541141270)

Abstract: *Services a system delivers are called dependable when it is trustworthy enough that reliance can be placed on them because they are available, reliable, safe and secure. In the contribution, overview of methods for reliability enhancement of embedded real-time systems is presented with focus on mechanisms related to fault and error diagnosis and on fault tolerancy of the systems. It presents actual trends as well as possible directions of further research activities related to corresponding areas.*

1. Introduction

Our previous research activities [9] [13] [14] [15] were focused on topics related to *diagnosis* and *testing* of digital systems. Primarily, we have dealt with *testability analysis* (TA) and its application to *design/synthesis for testability* (D/SFT) of digital systems. Our TA method was successfully taken as a basis of a cost/quality trade-off measure of solutions from a design search space. Principle of a D/SFT method based on our TA method could be described as follows: for a given *digital system* (given in a form of *net-list* and *library of components*), *design constraints* (e.g., maximal area and pin overheads, maximal power consumption for test application purposes) and set of *techniques* applicable for *testability enhancement* (partial/full scan, test-point insertion, built-in self test), the measure was used to evaluate each modification of original system by means of particular configuration of techniques. Over a set of possible modifications, *search-space exploration* algorithm (e.g., genetic algorithm) was used to find the highly-evaluated modification fulfilling design constraints maximally and having maximal testability value.

In the paper, the above-mentioned activities are put into a wider dependability context with focus to dependability of embedded systems controlled by a *real-time operating system* (RTOS). At the beginning, basic terms are explained together with state of the art in related areas. Then, principle of our approach and future research perspective are presented.

1.1 Dependability

Services a system delivers are called *dependable* when it is trustworthy enough that reliance can be placed on them because they are *available, reliable, safe, secure* etc. Because of width of the dependability related problems, the paper is focused only on mechanisms related to *fault/error diagnosis* and on *fault-tolerancy*. Further, the paper is restricted to *real-time embedded systems*.

1.2 Embedded Systems

According to [16] an embedded system is "*a data processing system which is built-in or embedded within a machine or a system. It partly or wholly controls the functionality and the operation of the machine. The data processing system and the enclosing system are dependent on each other in such a way that one cannot function without the other*". In previous two decades, the number of embedded systems in usage has grown at an enormous rate; this has been especially evident with the rise of cell phones, handheld computers, network and intelligent (e.g., automotive, home-appliance) devices. Flexible embedded systems are based on microcontroller (MCU) or FPGA technology. Many producers of those technologies exists (e.g., Freescale, Texas Instruments, Atmel, Xilinx, Altera) and offer their products in a wide range of parameters and architectures.

1.3 Real-Time Systems

On top of the above-mentioned, the paper is restricted to systems, whose logical correctness is based on both the correctness of outputs and timeliness of the outputs [3] [4] [5] [6] [7] . Such a system, i.e., that is able to produce a right response to given stimuli and the response is produced on time, is called a *real-time* (RT) *system*. Because also the class of embedded RT systems is too wide, we will focus only to systems driven by RTOS. Actually, many RTOSes of various parameters are available – e.g., LynxOS, OSEK/VDX, QNX, uC/OS-II, VxWorks, Windows CE. Also, many interfaces or modifications of existing, originally not-RT, products exist – e.g., POSIX, CORBA or Java RT extensions, Windows RTX extension etc.

2. Faults, Errors and Failures

A *fault* is a deviation in *hardware* (HW) or *software* (SW) component from its intended function [1] [7] . Faults can be classified in several ways, e.g., they can be classified by their duration (permanent, transient, intermittent faults), nature of output (malicious faults leading to Byzantine failure, non-malicious faults) or correlation to other faults (independent, correlated faults).

An *error* is a manifestation of a fault in a system, in which the logical state of an element differs from its intended value. If not detected, error is latent or it can disappear before it is detected. In many times, error propagates and creates other (new) errors. Usually, *related faults* manifest themselves as similar errors and lead to common-mode failures, whereas *independent faults* cause distinct errors and separate failures.

The time between fault occurrence and the first appearance of an error is called *fault latency*. The time between occurrence of an error and its detection is called *error latency*.

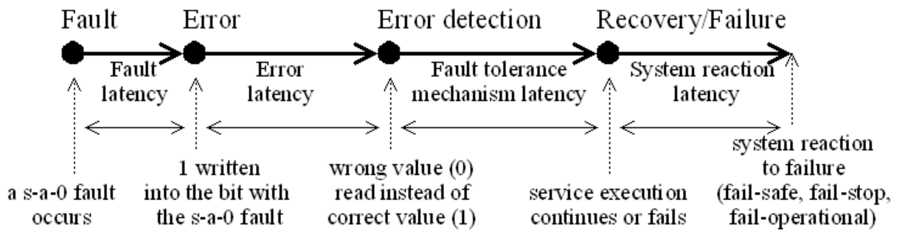
Fault
Error
Error detection
Recovery/Failure
Fault latency
Error latency
Fault tolerance mechanism latency
System reaction latency
a s-a-0 fault occurs
1 written into the bit with the s-a-0 fault
wrong value (0) read instead of correct value (1)
service execution continues or fails
system reaction to failure (fail-safe, fail-stop, fail-operational)

Fig. 1 – Illustration to Fault, Error and Latency terms

Many mechanisms exist for achieving dependability. Basically, they can be divided into following groups: *fault avoidance* (typically by construction), *fault removal* (t. by verification), *fault tolerance* (t. by redundancy) and *fault forecasting* (t. by estimation).

E.g., when a fault-tolerance mechanism detects an error, it may initiate several actions to handle the fault and contain its errors. Recovery occurs if these actions are successful; otherwise, the system eventually malfunctions and *failure* occurs.

In general, system can react in several ways to failures – e.g., *fail-safe* (system transits to a safe state), *fail-operational* (only subset of expected service is delivered by the system), *fail-stop* (system produces no output rather than incorrect one), or *no single point of failure* (it is ensured that a failure of a single component will not cause a failure of the system) approaches can be utilized during system design cycle.

3. Reliability Improvement Mechanisms: General Principles

It is evident that above-mentioned latencies can cause a significant problem if they are involved in RT systems because efforts of *error recovery mechanisms* can result in situation system reaction time is out of accepted interval because of the latencies. Thus, for RT systems, special mechanisms have to be designed able to perform recovery on time; otherwise, they can result in more or less serious system failure in spite of recovery itself was successful. In general, recovery mechanisms can be divided into two groups: *forward* (errors are corrected/masked without any computations to be redone) and *backward recovery* (system is rolled back to a believed pre-error state and performs all subsequent actions again) [5] [7] . Also, the mechanisms can be classified according to redundancy type they are based on: *HW/SW* (also called *spatial*) *redundancy, time redundancy, information redundancy*.

3.1 Information Redundancy

Methods belonging to this group are well known in general, so let us mention at least some of them: *data duplicity*, *parity*, *checksums* and *cyclic redundancy codes* (*CRC*s). This paper will not focus on those methods anymore.

3.2 Spatial Redundancy

Let spatial (HW/SW) redundancy techniques be presented in brief now. HW redundancy is usually realized by means of *N-modular redundancy* (*NMR*) and *voters*. Because it is very expensive (in the simplest case, supposing maximally *m* errors can appear in a component, (*m*+1) copies of the components must be implemented in the system), it is recommended to use it in extreme cases only. SW redundancy is usually based on one of following approaches [7] : *N-version programming* (concurrently running SW blocks, equivalent in functionality, but differing in implementation are used to detect an error, i.e., NMR analogy), *checkpoints* (immediate results of SW blocks are written to fixed memory places for diagnostics purposes) – see Fig. 2a, *recovery blocks* (at the end of each recovery block, checkpoints are tested for "reasonableness". If the result is not reasonable, the processing resumes with prior recovery block) – see Fig. 2b. Another approach to recovery blocks is presented in [5] – each recovery block is supposed to have *recovery memory* at its input and is able to process one of its code-variants (one of them is called *primary*, the others are called *secondary* or *backups*). Result of each variant (starting with the primary one) is checked by *acceptance test*. If the test fails, another variant is executed. If a correct result is produced by at least of one variant, it can be used by subsequent block(s). Otherwise, i.e., if all variants fail to produce a correct result, recovery attempt within the block fails (all recovery attempts are exhausted), which could be a signal to start the recovery in corresponding superior block.

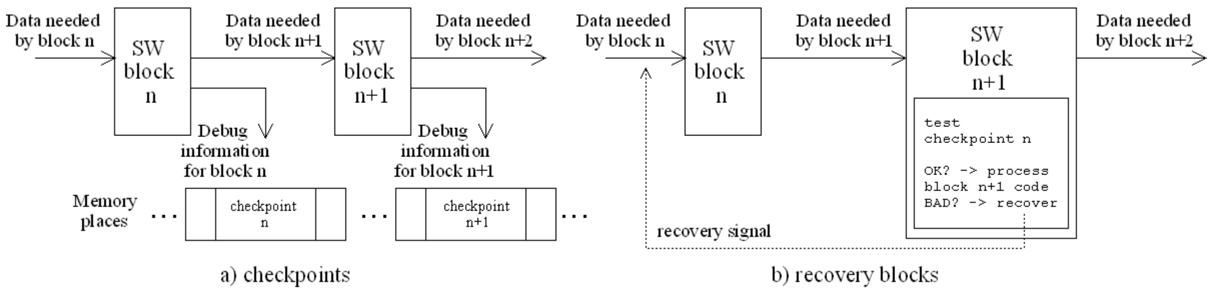

Fig. 2 – Illustration to Checkpoints and Recovery blocks

Reliability can be also increased by means of *software black boxes* (used to record in-system transitions for further analysis) or *built-in-test software* (*BITS*, used e.g. for ongoing diagnostics of underlying HW an embedded system runs on; for example, it can detect incorrectly working I/O channel, shut off it and redirect I/O to the properly working one). Because those methods, together with CPU, memory, internal block etc. diagnosis can consume much time than it is desired, it is recommended to process them in a *background mode* of an RT system.

3.3 Time Redundancy

Methods belonging to this group try to minimize price of spatial redundancy by moving recovery techniques to higher levels within system architecture – usually to system control layers. In principle, the methods are based on (re)*scheduling* computational units (HW/SW blocks) of the system (typically called *tasks*, each responsible for timely reaction to corresponding stimuli) in such a way the response to the stimuli is produced both in a correct way and on time. Each RT task can be described by means of several *static* (*r*: release time, *C*: worst-case execution time, *T*: invocation period, *D*: deadline etc.) and/or *dynamic parameters* (*R(t)*: response time, *L(t)*: laxity time etc.) used to evaluate task priority *P*, which can be static or dynamic depending on scheduling approach. So-called *scheduler* is responsible for those actions. It is a crucial part of an RTOS kernel and is responsible for assigning system resources (CPU, memory etc.) to particular tasks. Some of the approaches belonging to the group will be described in the next, as well as other approaches typically utilized for RT systems.

4. Reliability Improvement Mechanisms Applicable to Embedded RT Systems

According to [17] there are 3 kinds of errors that can appear during *task* execution: the *task* 1) enters a *dead-lock* state, in which it does not take any action, 2) produces *incorrect data* and 3) enters a *live-lock* state in which it fails to produce any result.

4.1 Watchdog Timers and Processors

Error 1) could be detected, e.g., by a *watchdog timer* (in correct function, task resets timer before it overflows; e.g., if task is in a dead-lock state, timer is not reset on time, so it overflows which is a signal for error detection mechanism) or *watchdog processor* (each task produces flags informing about actual position in its code; watchdog processor guards whether the *control flow* is correct or not and if it is incorrect, error is detected). After the error is detected, incorrectly functioning task could be reset and (re)scheduled again (of course, only if its reaction time will not exceed required limits).

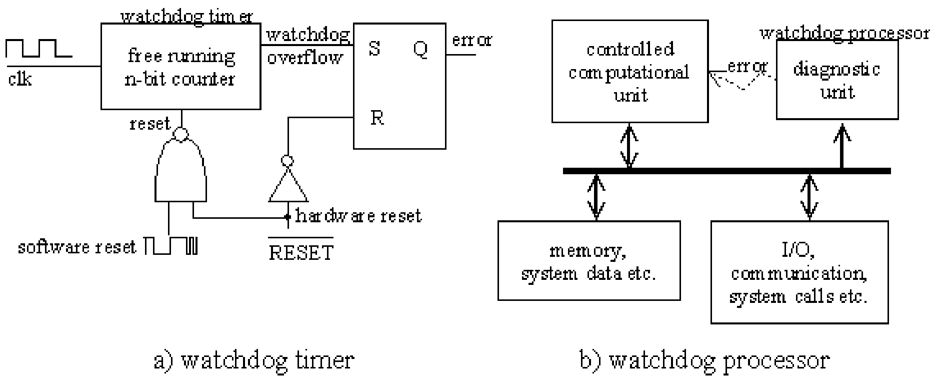

Fig. 3 – Illustration to Watchdog timers and processors

In case of errors 2) and 3) watchdog timers are serviced properly, so the errors remain undetected by means of them; thus, further techniques need to be utilized for their detection. E.g., to detect the error 3), each task can be designed in the following way: after a task invocation, output variables are set to “uninitialized value”, so it can be recognized at the end of task execution (or inner task observation points) that output value(s) were not changed by the task. In such a case, the error is detected. Thus, an *omission fault* can be converted to a *value fault* by the mechanism. Error 2) can be detected, e.g., by means of voter(s).

4.2 Scheduling Mechanisms

Any system has a finite processing power. If we intend to guarantee by design that certain performance requirements can be met, then we have to postulate a set of assumptions about the behavior of the environment: *load hypothesis* (defines the peak load that is assumed to be generated by the environment) and *fault hypothesis* (defines the types and frequency of faults that a system must be capable to handle). If a specified fault scenario develops, a system must still provide a specified level of service. If more faults are generated than what is specified in the fault hypothesis, then, sometimes, the performance of the system must *degrade gracefully*, i.e., the system must not suddenly collapse as the size of faults increases, rather it should continue to execute part of its workload.

Because classical RT scheduling approaches (RM/A, DM, EDF, LLF etc. [4]) are not designed to work in overload conditions, special scheduling mechanisms exist which are able to ensure a time-limited graceful degradation of an overloaded system in order to get its load *U* (defined as a sum of partial loads of particular tasks from a set τ of RT tasks) to normal limits. E.g., if U >= 100 % for a 1-CPU system, the system laxity is zero (i.e., the system is fully occupied with computations related to tasks actually present in the system), so if a new task arrives in this situation (i.e., stimuli from further input is to be processed by the system), 1) some of the *less important* task(s) could be suspended to let *more important* (*application critical*) tasks running (e.g., D_{over}, RED, DASA, LBESA mechanisms – see Fig. 4 for block schema) or 2) computations of some *less important* tasks could be switched, e.g., to a less-precise/fast-estimation mode (e.g., Statistical RM scheduling, Multi-frame, Elastic and On-line adaptive models, Deadline manipulation and Imprecise computation mechanisms [4]).

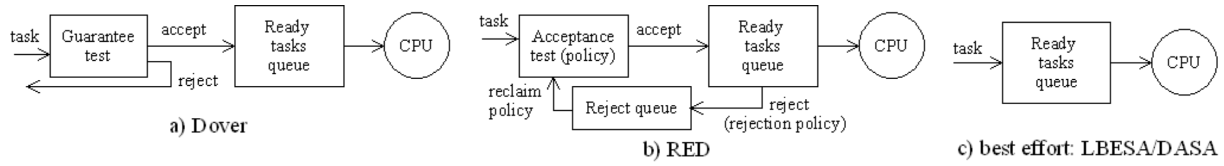

Fig. 4 – Illustration to selected scheduling mechanisms for overload conditions

5. Proposed Solution to Reliability of RT Systems

We have decided to take advantage of most of above-mentioned dependability improving mechanisms and to modify an existing wide-spread RTOS about those mechanisms. For practical implementation of the concepts, *uC/OS-II* that is available for more than 30 various embedded target platforms was selected [8] . Block schema of the proposed solution is depicted in Fig. 5.

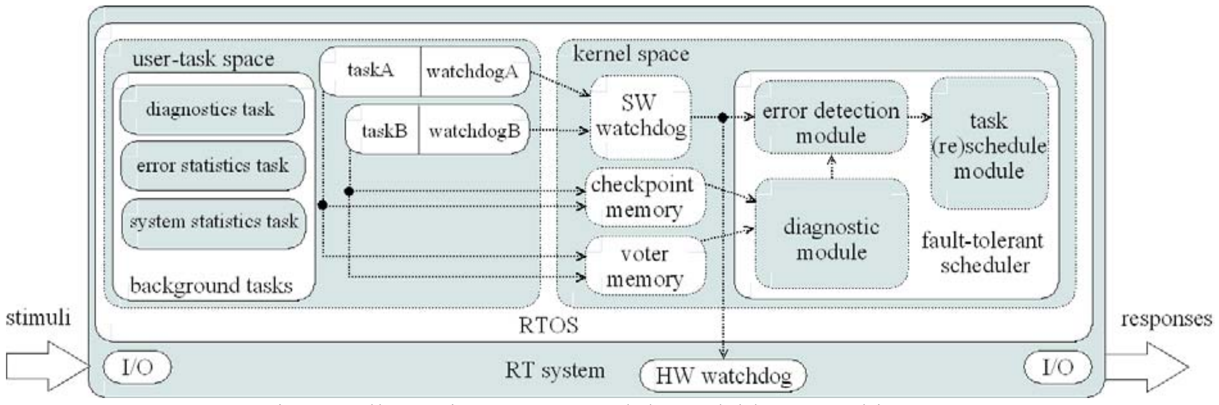

Fig. 5 – Illustration to proposed dependable RT architecture

Each task within a user-task space is assigned its own *separately configurable watchdog* (with a value stored in the user/kernel space controlled by a superior SW watchdog placed in the kernel space) able to detect a deadlock within the task. If the number of errors exceeds the acceptable level, HW watchdog is not serviced by SW watchdog properly, which could leads to *reset of a whole RT system* or switching the system e.g. to a *fail-stop/fail-safe state*. If the number of errors is in acceptable limits, error detection unit will try to (re)schedule malfunctioning tasks in order to mask or tolerate the errors in a *fail-operational state* of the system (including recovery block approach). Besides watchdog, each task contains a code able to update *checkpoint* data related to the task. The data is read and checked by a diagnostic unit (periodically in the ideal case) in order to detect a live-lock state of tasks. If detected, corresponding task can be (re)scheduled. *Incorrect data* produced by a task could be detected by means of a *voter memory* implemented in the kernel space and voter(s) explicitly implemented as a special task(s) or task portions in the user-task space or diagnostic unit part in the kernel space. The voter will produce a result after all data are prepared to make a decision. After done, the decision is processed by the error detection module. In order to be able to evaluate parameters like system load, task queue utilization, error frequency/type, special tasks (*diagnostics*, *error statistics*, *system statistics*, *maintenance* etc.) can be implemented in the background section of the user-task space. Because the tasks will be processed in system laxity time, they will not affect RT parameters of the system, but will contribute to dependability of the system.

6. Conclusion and Future Research Perspectives

In the paper, 1-CPU solution to the problem was discussed for simplicity reasons. However, the solution can be extended about more advanced concepts able to improve system dependability in a significant way. Essentially, there are two areas we plan to focus on: distributed and network system area [4] [6] [12] [16] (which is studied for a longer time) and partial dynamic reconfiguration area [10] (which belongs one of the most perspective nowadays research and engineering areas). Using the principles, malfunctioning computational units could migrate to different network/CPU node(s) or could, e.g., to repair or clone themselves in order to increase dependability of the system. But, this could probably lead to change of RT parameters of the system, so techniques like special buses presented in [11] should be used in order to minimize such unneeded changes.

Acknowledgement

The work related to the paper has been financially supported by the Research Plan No. MSM, 0021630528 – *Security-Oriented Research in Information Technology* and GAČR No. 102/09/1668 – *SoC Circuits Reliability and Availability Improvement*.

[1] Bushnell, M. L., Agrawal, V. D.: *Essential of Electronic Testing for Digital, Memory & Mixed-Signal Circuits*. Springer, 2000, 712 p., ISBN 0-7923-7991-8.

[2] Chakravarty S., Tomar, R., Arora, M.: *Need a watchdog for improved system fault tolerance?* Available at <http://www.embedded.com/design/211600220>, accessed on March 16, 2009.

[3] Cheng, A. M. K.: *Real-Time Systems: Scheduling, Analysis, and Verification*. Wiley, 2002, 552 p., ISBN 0-471-18406-3.

[4] Cottet, F., Delacroix, J., Kaiser, C., Mammeri, Z.: *Scheduling in Real-Time Systems*. John Wiley & Sons, 2002, 266 p., ISBN 0-470-84766-2.

[5] Čeleda, P.: *Zvýšení spolehlivosti a diagnostika operačních systémů pracujících v reálném čase*. Ph. D. Thesis, Defence University, Brno, 2007, 122 p.

[6] Kandasamy N., Hayes, J. P., Murray, B. T.: *Time-Constrained Failure Diagnosis in Distributed Embedded Systems: Application to Actuator Diagnosis*. IEEE Transactions on Parallel and Distributed Systems, 16(3), pp. 258-270.

[7] Laplante, P. A.: *Real-Time Systems Design and Analysis*. Wiley-IEEE Press, 2004, 528 p., ISBN 0-471-22855-9.

[8] Micrium: *Micrium.com: Embedded Software Components*. Available on-line at <http://www.micrium.com>, accessed on March 23, 2009.

[9] Pečenka, T., Kotásek, Z., Sekanina, L., Strnadel, J.: *Automatic Discovery of RTL Benchmark Circuits with Predefined Testability Properties*, In: Proc. of the 2005 NASA/DoD Conference on Evolvable Hardware, Los Alamitos, ICSP, 2005, pp. 51-58, ISBN 0-7695-2399-4.

[10] Rupe, D., Kenny, J., R.: *Two Competitive FPGA Methodologies for Run-Time Reconfiguration*. Technology Feature. 4 p., 2008.

[11] Rushby, J.: *A Comparison of Bus Architectures for Safety-Critical Embedded Systems*. NASA/CR-2003-212161 Contractor Report, 63 p., 2003.

[12] Rubel, P., Gillen, M., Loyall, J., Gokhale, A., Balasubramanian, J., Paulos, A., Narasimhan, P., Schantz, R.: *Fault-Tolerant Approaches for Distributed Real-Time and Embedded Systems.* In: Proceedings of Military Communication Conference, 2007, 8 p. ISBN 1-4244-1513-06.

[13] Strnadel, J.: *Testability Analysis and Improvements of Register-Transfer Level Digital Circuits*, In: Computing and Informatics, 25(5), 2006, Bratislava, pp. 441-464, ISSN 1335-9150.

[14] Strnadel, J.: *TASTE: Testability Analysis Engine and Opened Libraries for Digital Data Path*, In: Proceedings of 11th Euromicro Conference on Digital Systems Design Architectures, Methods and Tools, Los Alamitos, IEEE CS, 2008, pp. 865-872, ISBN 978-0-7695-3277-6.

[15] Strnadel, J., Pečenka, T., Kotásek, Z.: *Measuring Design for Testability Tool Effectiveness by Means of FITTest_BENCH06 Benchmark Circuits*, In: Computing and Informatics, Vol. 27, No. 6, 2008, Bratislava, pp. 913-930, ISSN 1335-9150.

[16] Townend, P., Xu, J., Munro, M.: *Building Embedded Fault-Tolerant Systems for Critical Applications: An Experimental Study*, In: Distributed and Parallel Embedded Systems, Kluwer, 2002, 10 p.

[17] Zhang, M., Liu Z., Ravn, A. P.: *Design and Verification of a Fault-Tolerant System*.