Task-Level Modeling and Design of Components for Construction of Dependable Time-Critical Systems Implemented by Means of RT Kernel

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: *In the contribution, the approach to modeling and design of components for the construction of dependable time-critical systems implemented by means of RT kernel is presented. On top of that, faults and errors in components and RT kernels are classified in the contribution. It is shown how safety of the components can be formally verified by means of the UPPAAL tool. Reliability is solved by means of time and spatial redundancies, both implemented on RT task scheduling level. The proposed solution is analyzed from view of schedulability of the resulting time-redundant set of RT tasks and its implementation is demonstrated by means of the uC/OS-II kernel.*

1. Introduction

Services a system delivers are called *dependable* when it is trustworthy enough that reliance can be placed on them because they are *testable, available, reliable, safe, secure* [3] etc. In Fig. 1, it is shown if problems related to dependability are taken into account in early phases of system's life-cycle then total redundancy (i.e., costs for dependability) can be minimized – or, at least close to minimum.

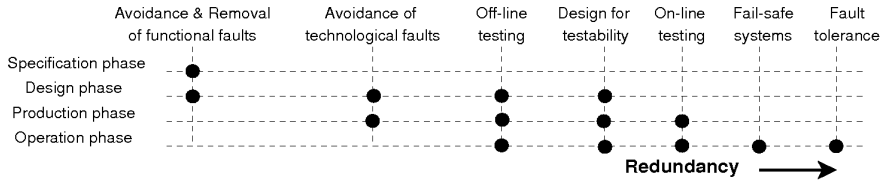


Fig. 1 – Typical Techniques and Application Areas Related to Dependability

In the past, we have dealt mainly with problems related to *testability* of digital systems [8] [12] [15] [16] [17] – at our faculty, we have developed complex set of tools and methods, which can be utilized to maximize testability of the digital system under given design constraints including area/pin overheads, power dissipation etc. It can be said the techniques related to testability are typically applied at lower levels than the techniques related to the attributes. It is because they strongly depend on the hardware the system is running on. Their advantage can be seen in the fact they are able to detect[1] a *fault*[2] (i.e., a *failure*[3] case) before it possibly manifests itself as an *error* in higher levels of the system's hierarchy. This implies the levels must be able to deal with the errors. Actually, our activities are focused to further dependability attributes – especially to *availability*, *reliability* and *safety*. There are many techniques, which can be utilized to contain the fault effects. Traditionally, the following groups are distinguished: *fault avoidance* (typically by construction), *fault removal* (t. by verification), *fault tolerance* (t. by redundancy) and *fault forecasting* (t. by estimation) techniques. The paper is focused mainly to fault removal/tolerance techniques based on recovery mechanisms and combination of time and spatial redundancies. Proper combination of the techniques must be selected if critical systems are developed as their failures can have dramatic consequences [3] .

Because of limited scope of the paper, the paper is focused only to the techniques applicable during design of time-critical systems – each system that is able to produce the right response to the given stimuli and the response is produced on time, is called the *real-time* (RT) *system*. Typically, RT systems are controlled by means of a *real-time operating system* (RTOS)[4], so the techniques presented in the paper are put just into the RTOS context.

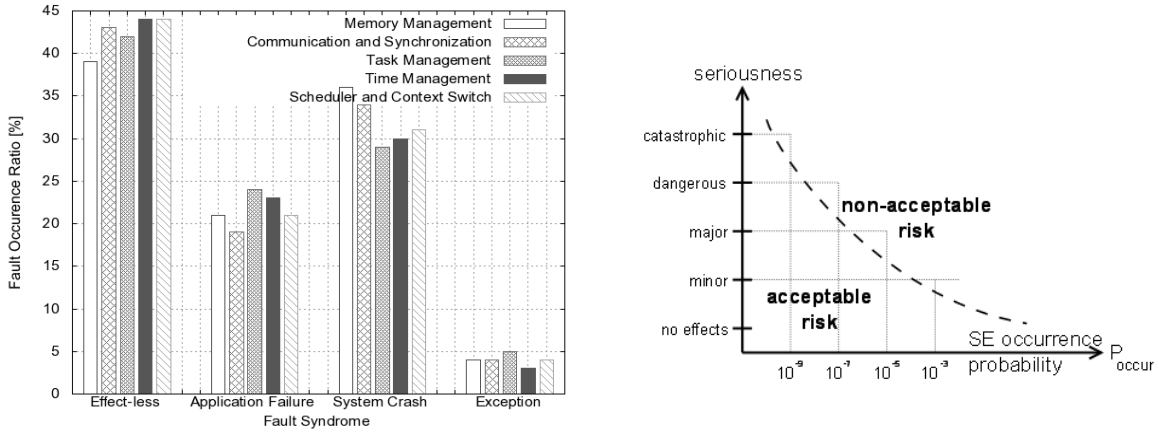
---

[1] or, some of them also to locate
[2] i.e., a deviation of the component from its intended function
[3] failure characterizes a wrong service delivered by a system, i.e., behavior that is not in compliance with the expected one
[4] also called an RT kernel

1.1 Classification of Faults

According to [13] , there are many criteria which can be utilized to classify the faults. Based on *duration*, faults can be classified as a) *transient* or b) *permanent*: a transient fault will eventually disappear without any apparent intervention, whereas a permanent one will remain unless it is removed. It may seem permanent faults are more severe, but they are much easier to diagnose and handle. A particularly problematic type of (transient) fault is the c) *intermittent* fault that recurs, often unpredictably. A different way to classify faults is by their underlying *cause*: a) *design faults* are the result of design failures while b) *operational f.* occurring during the lifetime of the system. Finally, based on how a failed component behaves once it has failed, faults can be classified into the following categories: a) *crash faults* – the component either completely stops operating or never returns to a valid state, b) *omission f.* – the component fails to perform its service, c) *value f.* – the component produces a wrong data, d) *liveness f.* – the component deadlocks e) *timing f.* – the component does not complete its service on time etc.

Systems are subject to different parasitic phenomena and faults induced by the environment (e.g., *single event upsets*, *SEU*s). The *soft-errors*[5] (*SE*s) they cause can lead to incorrect system behavior – typically because of unintended change of bit within the memory element (e.g., RAM cell, CPU register). So, the RTOS services must be robust enough if the system is to continue even in the presence of faults. SE faults can cause several syndromes when RT kernel services are corrupted. In [6] , the syndromes are classified as follows: a) *effect-less* – no visible effect on the system behavior is observed, b) *exception trigger* – the system triggers an exception routine (e.g., illegal instruction, division-by-zero), c) *system crash* – the system stops functioning, d) *application-level failure* – fault consequences propagate to the application level (the following fault subclasses can be distinguished: *crash/omission/value/timing faults* etc.) – see Fig. 2a. In the figure, it can be seen that almost 50 % of SEs have no effect to RT system behavior. But on the other hand, remaining 50 % of SEs leads to serious failures, which manifest themselves at higher levels within the RT system structure.



(a) (b)

Fig. 2 – Illustration to (a) Syndromes of SE Faults [6] and (b) Seriousness Classes [3]

1.2 Classification of Failures

Failures altering behavior of the product can be grouped together into *seriousness classes* according to the seriousness of their consequences [3] . The classes can be formed on basis of a quantitative evaluation of the consequences (e.g., class containing 0 causalities, class with 1 to 3 causalities etc.) or, a qualitative evaluation, e.g., to a) *catastrophic* (*disastrous*) leading to human live loss and to wide devastation of system's environment, b) *dangerous* (*serious*) leading to a small number of serious causalities and injuries, c) *major* (*insignificant*) leading to light injuries and/or causalities, d) *minor* (*benign*) making people upset because of discomfort, e) *without effect*. For each failure, probability of its occurrence (P_{occur}) can be determined. In relation to this, classes can be defined according to value domains of the failure probability, e.g., a failure can be said a) *probable* if its $10^{-5} < P_{occur}$, b) *rare* if $10^{-7} < P_{occur} \leq 10^{-5}$, c) *extremely rare* if $10^{-9} < P_{occur} \leq 10^{-7}$, d) *extremely*

[5] it is called „soft“ because the upset memory element remains operational and able to eventually store new information when a write operation on that same element is performed

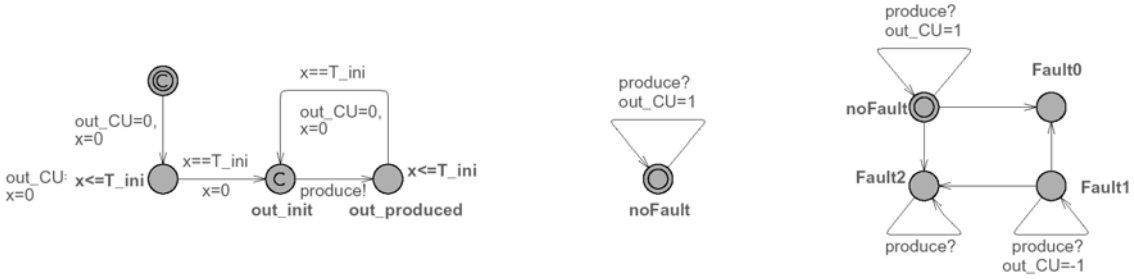
improbable if $P_{occur} \leq 10^{-9}$). If a failure seriousness class is associated with an *acceptable risk rate* (i.e., the maximum probability of the failures acceptable for the given seriousness class), *safety class* can be defined – illustration to the safety classes can be seen in Fig. 2b together with so-called *acceptability curve* (dashed line).

1.3 Classification of Responses and of RT Systems

According to seriousness of failure consequences, RT systems and responses produced by the RT systems can be classified. Typically, following types are distinguished [1] [2] [9] : a) *hard*: if the only one deadline is not met, the impacts are catastrophic, permanent and irreversible – a lot of people and property are affected (e.g., nuclear reactor explosion), b) *firm*: deadlines must be met within specified tolerance interval; otherwise, the impacts could be serious, but limited only to people and property within an immediate environment (e.g., insuline pump or airbag control failure), c) *soft*: if deadlines are not met, quality of services delivered by a system is temporarily reduced; the impacts to environment are negligible (e.g., missing a deadline within video-game engine).

2. Modeling of Dependable Time-Critical Components

If a system is required to be dependable, it must be composed of dependable components. Thus a special attention must be paid to design of components the system consists of. In the subsection, approach to design safe components is outlined. A component can be seen as a *computational unit* (*CU*) processing its inputs (in_{CU}) and producing $f_{CU}(in_{CU})$ at its output (out_{CU}). In general, a CU can behave in one of the following ways [18] : a) a *correct value*, i.e., $out_{CU} = f_{CU}(in_{CU})$ is produced at the output, b) an *incorrect value*, i.e., $out_{CU} \neq f_{CU}(in_{CU})$ is produced at the output, c) *no value* is produced at the output. In order to model the behavior, output of each CU must be initialized at the beginning of an initialization period (T_{ini}). This can be done, e.g., by the *synchronization unit* (*SU*). After out_{CU} is initialized, SU waits for T_{ini} time units and then sends a signal to CU to force it to read its inputs and compute and produce the output. In the model, the correct output is represented by 1, incorrect by -1 and initialized by 0. To model the behavior more precisely, we have extended SU model presented in [18] – the extended model can be seen in Fig. 3a. Models of fault-free (faulty) CUs are unchanged comparing to [18] – see Fig. 3b (Fig. 3c).



a) SU b) fault-free CU c) faulty CU

Fig. 3 – The Blocks for Modeling of Fault-Tolerant Components

A fault-free system composed of CU and SU satisfies the following properties [18] : a) every period of T_{ini} time units the CU computes the (new) correct value and produces it at out_{CU}, b) the output value is kept for T_{ini} time units before it is cleared.

In the *faulty CU* model (see Fig. 3c), the following type of faults are considered: a) *FAULT0*: the CU enters a *deadlock* state in which it does not take any action, b) *FAULT1*: the CU produces incorrect data (*a value fault*), c) *FAULT2*: the CU enters a *live-lock* state (*an omission fault*). While errors caused by faults of FAULT0 type can be detected, e.g., by means of a *watchdog* (*WDG*), the others must be detected by different mechanism because the CU resets the WDG properly in the FAULT0 case.

The problems related to detection of the errors and consequent recovery from the errors in RT systems are typically solved, e.g., by means of scheduling mechanisms designed especially for the purposes (*Fault-Tolerant Rate Monotonic Mechanism*: *FT-RMS* [4] , *Imprecise Computation FT-RMS*: *IC-FT-RMS* [7] etc.), of WDG mechanisms, of "classical" spatial/time/information redundant techniques [3] or of *control-flow checking* techniques [5] [11] . Each of the techniques can solve the problem at different levels within the RT system –computational resources layer, hardware abstraction

layer, RT kernel level, RT task level, etc. In Fig. 2a, it can be seen that approximately 25 % of errors propagate to the application layer and more than 30 % of errors lead to crash of the system.

3. Design of Dependable Time-Critical Components

So, we have decided to spread the dependability mechanisms into more layers in order to both detect the errors and to activate recovery mechanism on-time. 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: a) *load hypothesis*[6] and *fault hypothesis*[7]. If a specified fault scenario develops, the system is expected to provide the specified level of service. If more faults are generated than it is specified in the fault hypothesis then the performance of the system should be designed to *degrade gracefully*[8] rather than continue to execute part of its workload.

Actually, our research is concentrated to the design and consecutive implementation of a generic fault-tolerant architecture applicable to RT system implementations controlled by means of an RTOS. For the implementation, we have chosen *uC/OS-II* [10] RT kernel, which is a portable, scalable, multitasking, preemptive, deterministic, robust and reliable RT kernel. Because of its properties, it was certified for the use in safety-critical systems (in 2000 by the *Federal Aviation Administration – FAA*).

3.1 Characteristics of the Proposed Architecture

Each RT task (τ) is assigned one periodic *WDG task* (w_τ) that is of a higher priority. w_τ must be invoked as soon as possible in τ's body in order to be ready to detect an error and to activate appropriate recovery mechanism on-time. If it is ready, w_τ waits for a task-specific number of time units for the message incoming from τ. If the message is received before the w_τ overflows then w_τ is reset, τ is checked for omission/value faults and the next w_τ deadline is adjusted (Fig. 4, steps 1-4). If the fault is detected or w_τ is not reset on-time, the recovery mechanism is initiated (Fig. 4, steps 7-10).

Along with w_τ each τ is assigned the checkpoint memory block (the recovery memory block) used to store temporary results produced by τ (and the last known error-free state of τ).

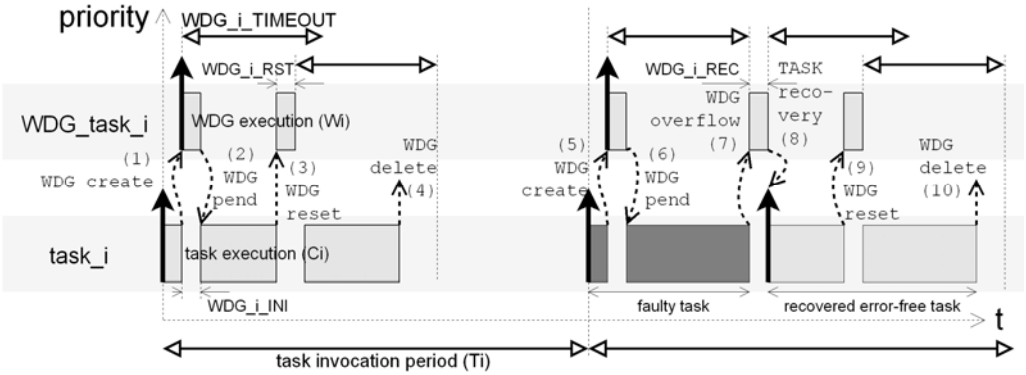


Fig. 4 – An Illustration to Timings Related to Error-Free and Faulty Tasks

Except w_τs the RT system periodically executes the *system watchdog* task (*SYS WDG*) responsible for gathering data about meeting reset deadlines related to particular w_τs and about correctness of the data stored in checkpoint and recovery blocks related to the tasks. Because there is no observer over the *SYS WDG*, the *SYS WDG* must be robust enough to resist to the given level of SEUs – least, it is recommended to equip the *SYS WDG* with the combination of control-flow checking and information redundancy techniques.

If levels of detected faults and missed deadlines detected by the *SYS WDG* are within an acceptable range[9], *hardware watchdog* (*HW WDG*) is reset on-time and the *SYS WDG* try to (re)schedule the malfunctioning tasks to mask/tolerate the errors in a fail-operational state of the system. If unsuccessful, the system can be switched to fail-stop or fail-safe state or hardware can be reset by the *HW WDG*. Recall that along with the hardware the entire RT system is reset; this leads to the extra

[6] defines the peak load that is assumed to be generated by the environment
[7] defines the types and frequency of faults that a system must be capable to handle
[8] the system must not suddenly collapse as the size of faults increases
[9] i.e. – despite of the situation, the deadlines are guaranteed by the scheduling mechanism

response latency and possibly to the loss of stimuli which have arisen during the reset process. However, in some situations the hardware reset is the only way how to return the system to the (initial or the last-known) error-free state in a relatively short time.

The basic blocks related to the proposed architecture and their dependences are depicted in Fig. 5.

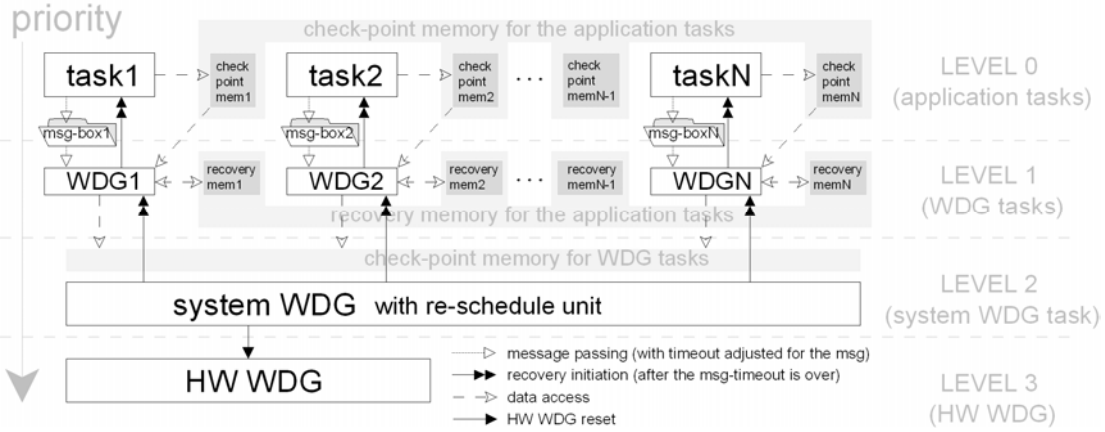


Fig. 5 – An Illustration to the Proposed Architecture

The special attention must be paid to the setup of parameters related to particular τs and w_τs – the relations (1) and (2) among the parameters must be fulfilled to guarantee that all deadlines of the RT tasks will be met during the run-time of the RT system. Symbols of following meanings are utilized in the relations: a) C_i denotes the worst-case execution time of τ_i, b) *WDG_i_INI* denotes the time needed to initialize the w_π task, c) *WDG_i_RST* denotes the deadline for resetting the w_τ task , d) N_{res} (N_{rec}) denotes the maximal number of w_π resets (maximal number of recovery tasks) which can occur during T_i (i.e., during the invocation period of τ_i), b) Δ_i denotes the maximum time for which the execution of τ_i is delayed[10], and e) $C_{lp}(\tau_i)$ denotes the time needed to execute tasks with priorities lower than P_{τ_i}.

$$C_i^* \overset{def}{=} (C_i + WDG\_i\_INI + N_{res} \times WDG\_i\_RST + \Delta_i) \times (N_{rec} + 1) \le T_i \quad (1)$$

$$C_{lp}(\tau_i) + C_i^* \le T_i \quad (2)$$

The skeleton of the possible implementation of the LEVEL-0 and the LEVEL-1 parts of the proposed architecture can be seen in Fig. 6.

```
01 static void task_n_body (void *p_arg)
02 {
03    OSTaskCreate(task_n_watchdog,     /* WDG task
creation */
04       (void *)0,
05       (OS_STK *)&task_n_wdg_stk[TASK_WDG_STK_SIZE-1],
06       TASK_n_WATCHDOG_PRIO);
07
08    for(;;) /* -- code of the task -- */
09    {
10      /* init data */
11      /* perform function */
12      /* create check & restoration points */
13      /* produce output */
14      OSTimeDly(TASK_n_PERIOD); /* wait for new period
*/
15    }
16
17    OSTaskDel(TASK_n_WATCHDOG_PRIO);  /* WDG deletion */
18 }
```

```
01 static void task_n_watchdog (void * p_arg)
02 {
03     INT8U err;
04     void *msg;
05
06     for(;;)
07     {
08        msg = OSMboxPend(msg_WDG_n, WDG_n_TIMEOUT,
&err);
09        if (msg != (void *)0) /* -- WDG reset on-time -
- */
10        {   /* check for omission and value faults
11        }      then reset SYS_WDG */
12        else
13        { /* -- WDG timeout over -- */
14          if (msg != (void *)0)
15          { /* initiate recovery mechanism */ }
16        }
17     }
18 }
```

Fig. 6 – An Illustration to the Implementation of τ and w_τ by the Means of the uC/OS-II kernel

4. Conclusion

The single-CPU approach to the dependability enhancements of RT systems at the task-level was presented in the paper. Because of limited computational resources available in the single-CPU environment, it cannot be guaranteed that there will be enough CPU time for recovery of each malfunctioning task if the relations (1) and (2) are not fulfilled.

Main disadvantage of the method can be seen in the fact that for N application tasks, it needs N+1 extra RT tasks (N watchdog tasks plus 1 system watchdog tasks) plus extra memory blocks plus

[10] e.g., by higher-priority tasks, blocking shared resources, RT kernel delays

overheads related to communication among the components within the architecture. The presence of the extra tasks in the system can cause deadlines of the other tasks are missed; in this situation a proper trade-off among dependability level and deadline-meeting level must be done.

Further disadvantage can be seen in the fact tasks must be processed in a sequence (even though they seem to run in pseudo-parallel because of preemptive scheduling). In the future, we plan to remove the drawback by extending the approach about instruments typically utilized in distributed/network design and partial dynamic reconfiguration 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 overall dependability of the system. However, it is a generally known fact the problem cannot be simply solved, e.g., by adding "few extra" CPUs into the system – again, such solutions must deal with anomalies related to increasing of computational resources, weakening timing constraints etc., which needs further problems must be dealt with.

Acknowledgements

The research related to the paper was supported by the Grant Agency of the Czech Republic (GACR) No. 102/09/1668 – "SoC circuits reliability and availability improvement", by the grant "BUT FIT-S-10-1" and by Research Project No. MSM 0021630528 – "Security-Oriented Research in Information Technology".

References

[1] Cheng, A. M. K.: *Real-Time Systems: Scheduling, Analysis, and Verification*. Wiley, 2002, 552 p.

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

[3] Geffroy, J.-C., Motet, G.: *Design of Dependable Computing Systems*. Kluwer Academic Publ., 2002, 692 p.

[4] Ghosh, S., Melhem, R., Mosse, D., Sarma, J.: *Fault-Tolerant Rate Monotonic Scheduling*. Journal of Real-Time Systems, Vol. 15, No. 2, 1998

[5] Goloubeva, O., Rebaudenko, M., Sonza Reorda, M., Vilolante, M.: *Soft-Error Detection Using Control Flow Assertions*. In: Proceedings of IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems, 2003, pp. 581-588

[6] Ignat, N., Nicolescu, B., Samaria, Y., Nicolescu, G.: *Soft-Error Classification and Impact Analysis on Real-Time Operating Systems*. In: Proc. of Design, Automation and Test in Europe (DATE), 2006, pp. 182–187

[7] Liu, J.W.S., Shih, W.-K., Lin, K.-J., Bettati, R., Chung, J.-Y.: *Imprecise Computations*, Proceedings of the IEEE, Vol. 82, No. 1, 1994, pp.83-93

[8] Kotásek, Z., Škarvada, J., Strnadel, J.: *Reduction of Power Dissipation Through Parallel Optimization of Test Vector and Scan Register Sequences*. Accepted for publication at the 13th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010, 6 p.

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

[10] Micrium: *Embedded Software Components*. Available on-line at <http://www.micrium.com>.

[11] Oh, N., Shirvani. P. P., McCluskey, E. J.: *Control Flow Checking by Software Signatures.* IEEE Transactions on Center for Reliable Computing, Technical Report, Vol. 51, No. 1, 2002, pp. 111-122

[12] 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.

[13] Selic, B.: *Fault tolerance techniques for distributed systems*. c2004. Available on-line at <http://www.ibm.com/developerworks/rational/library/114.html>.

[14] 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.

[15] 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.

[16] 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, pp. 913-930.

[17] Strnadel J.: *TASTE: Testability Analysis SuiTE*. c2008. Available on-line at <http://www.fit.vutbr.cz/~strnadel/diag/taste.htm>.

[18] Zhang, M., Liu, Z., Morisset, C., Ravn, A.: *Design and Verification of Fault-Tolerant Components*. In: Methods, Models and Tools for Fault Tolerance, Springer, 2009, pp. 57–84.