Non-Intrusive System Level Fault-Tolerance

Kristina Lundqvist, Jayakanth Srinivasan, Sébastien Gorelov

Embedded Systems Laboratory, Department of Aeronautics and Astronautics
Massachusetts Institute of Technology, Cambridge, MA 02139
{kristina, jksrini, gorelov}@mit.edu

Abstract. High-integrity embedded systems operate in multiple modes, in order to ensure system availability in the face of faults. Unanticipated state-dependent faults that remain in software after system design and development behave like hardware transient faults: they appear, do the damage and disappear. The conventional approach used for handling task overruns caused by transient faults is to use a single recovery task that implements minimal functionality. This approach provides limited availability and should be used as a last resort in order to keep the system online. Traditional fault detection approaches are often intrusive in that they consume processor resources in order to monitor system behavior. This paper presents a novel approach for fault-monitoring by leveraging the Ravenscar profile, model-checking and a system-on-chip implementation of both the kernel and an execution time monitor. System fault-tolerance is provided through a hierarchical set of operational modes that are based on timing behavior violations of individual tasks within the application. The approach is illustrated through a simple case study of a generic navigation system.

Introduction

Embedded systems are becoming permeating every facet of our daily lives, ranging from the control of toasters to managing complex flight control operations. A crucial segment of the embedded systems market addresses the needs of high-integrity systems, i.e., systems whose incorrect operation leads to significant losses in monetary terms, in terms of human lives or a combination thereof. High-integrity embedded real-time systems have to address the requirements imposed by the need for high-integrity as well as to satisfy the real-time nature of the system. By real-time, we mean the need to operate within the temporal constraints on system behavior. There are a number of well proven approaches for developing predictable real-time systems, in which the correctness of temporal behavior is assured in a systematic manner [11, 12]. A good example is fixed priority scheduling, which assumes that the system is formed by a fixed-set of tasks that provide system capabilities through periodic/aperiodic execution. This approach works extremely well when the system operates in a single mode, however, the class of systems, Figure 1., addressed in this paper display multi-moded behavior i.e. the system has a set of modes that involve overlapping sets of tasks providing different capabilities depending on the current state of the system, and the environment in which the system operates.

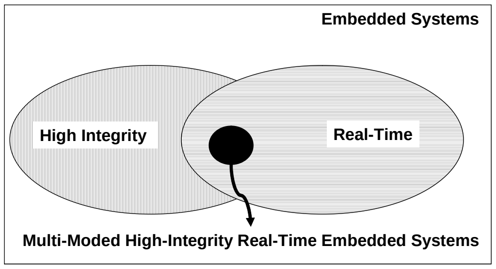


Fig. 1. Embedded Systems Problem Space

These systems undergo significant verification and validation activities prior to fielding. However, there are unanticipated state-dependent faults that remain in operational software after system design and development. These faults behave like hardware transient faults: they appear, do the damage and disappear [17]. In the context of real-time systems, the most visible manifestation of this class of faults is tasks missing their deadlines either through overruns or underruns. A multi-moded high-integrity real-time embedded system has to have the ability to detect the violation of timing bounds, and transition to the appropriate operational mode, while retaining predictable behavior and providing continued service. This paper presents a novel approach that leverages formal methods, System-on-chip design and the Ravenscar profile [2] to provide non-intrusive system level fault-tolerance.

The remainder of the paper is organized as follows: The *Technology* section discusses the three key areas that enable our approach for non-intrusive monitoring, and allow mode-transitions to provide continued service when tasks violate their timing bounds; the section on *Approach* provides an overview of the modeling, analysis and implementation adopted in this paper; the *Gurkh Generic Navigation System* section illustrates the approach using a simple case study of a navigation system; the *Conclusions* section documents the limitations of the current approach and charts the path forward in terms of future work.

Technology

Three key technologies have enabled us to reconsider and challenge the conventional approach of handling timing overruns of tasks. The first is the Ravenscar Profile [2, 4], a subset of the Ada 95 tasking model, which allows for analyzable deterministic concurrent tasking. The second is the emergence of low-cost system-on-chip technologies that contain embedded processors and Field-Programmable Gate Arrays (FPGAs). The third is the successful development and use of model-checking tools, e.g., UPPAAL [6], to automate the formal verification.

The Ravenscar Profile of Ada 95

In the domain of high-integrity real-time embedded systems, the use of Ada 83 run-time features, such as the rendezvous mechanism, select statements, and abort statement make deterministic analysis of the application infeasible [2]. The non-determinism and potentially blocking behavior of tasking or run-time calls when these features are used makes it impossible to derive an upper bound on execution time, which is critical for schedulability analysis. The Ravenscar Profile [2, 4] defines a subset of the Ada 95 tasking model to meet the requirements for determinism, schedulability analysis, and memory-boundedness associated with high-integrity real-time embedded systems. Additionally, the profile enables the creation of a small and efficient run-time system that supports task communication and synchronization. The Ravenscar Profile mandates the use of a static task set in the system and only allows inter-task communication to occur via protected objects. A static task set implies that the system has a fixed number of tasks at all time, hence the tasks cannot be created dynamically or terminate. Tasks have a single invocation event, but can have potentially unbounded number of invocations. Task invocations can either be time-triggered (tasks executing in response to a time event, such as delays) or event-triggered (executing in response to an event external to tasks). Task scheduling is carried out in a pre-emptive highest priority first manner. These restrictions imposed by the Ravenscar Profile allow systems to be analyzed for both functional and timing behavior.

Xilinx Virtex II Pro Platform

The Xilinx Virtex II Pro platform [18] contains an embedded PowerPC (PPC) core and an FPGA. The complete system architecture is shown in Figure 2. The software component of the system is implemented as a set of Ada 95 tasks that run on the PPC. A hardware implemented runtime kernel called RavenHaRT provides inter-task communication and scheduling services.

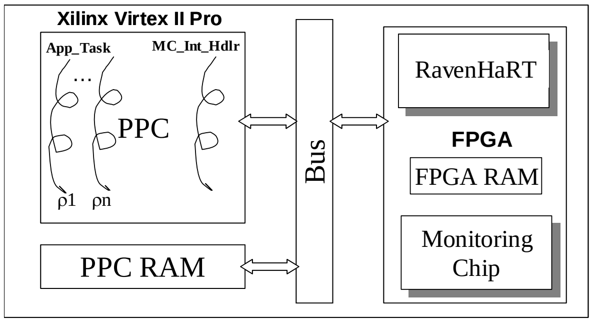


Fig. 2. System Architecture

RavenHaRT [16] is a formally verified, deterministic run-time system based on the Ravenscar Profile. The kernel specification enables the user to create a custom run-time system that can be synthesized onto the FPGA at the end of system design. The other critical component is the monitoring chip (MC). The MC contains a set of execution timers associated with application tasks, and is synthesized onto the FPGA. The MC monitors the timing behavior based on information from RavenHaRT, and informs RavenHaRT when a timing violation is detected.

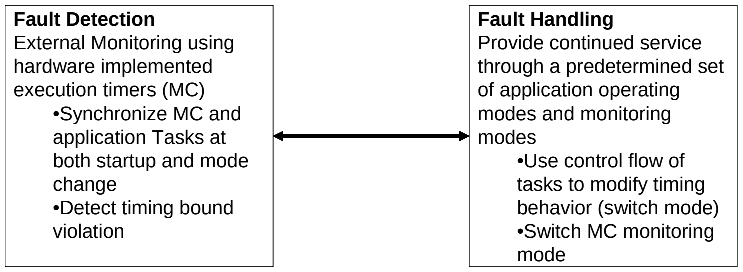
Model Checking using UPPAAL

The UPPAAL model-checking toolkit [4, 6] consists of three main parts: a description language, a simulator and a model checker. The idea behind the tool is to model a system using the graphical user interface and timed automata [1, 8], validate the system by simulation, and finally verify the system that it is correct with respect to a set of properties. UPPAAL uses a non-deterministic guarded command language to describe the system behavior as networks of automata extended with clock and data variables. The simulator is a validation tool, which can be used to examine a set of possible dynamic executions of the system as part of the design process. The model checker uses (directed) state space exploration to cover the dynamic behavior of the system and check invariant and bounded-liveness properties

Approach

The intuitive principle behind implementing fault-tolerance in a system is to increase design robustness by adding redundant resources and the mechanisms necessary to make use of them when needed [10]. Fault-tolerance mechanisms can be broadly partitioned into fault detection and fault handling. Fault detection mechanisms identify the occurrence of the fault and determine when to initiate/trigger a recovery action. The fault handling mechanisms act on the signal provided by the fault detection mechanism to protect the system either by reconfiguration of resources or by transitioning to a safe mode.

Conventionally high-integrity real-time embedded systems are built using cyclic schedulers [4]. A task exceeding its budgeted execution time over a cyclic schedule can be easily detected, and necessary corrective action can be taken. This is carried out by checking if the current action was completed by the task when a minor cycle interrupt occurs. If the action has not been completed, then it is assumed that a task has overrun its budgeted time, and the necessary fault-handling mechanism is adopted. Preemptive multi-tasking schedulers make system design a lot simpler through the use of concurrency, but there are no comparable approaches for detecting and handling execution time overruns [7]. Classical overrun management schemes that use techniques such as dynamic priorities and aborts are not Ravenscar compliant. Work carried out by de la Puente and Zamorano proposes a Ravenscar compliant scheme that allows a supervisory task to detect overruns and preempt the faulty task [5]. Similar work carried out by Harbour et.al, [7], proposes an execution time clock library that can be used to monitor timing behavior of the executing application. Both approaches are constrained by the fact that the monitor itself alters the timing behavior of the total system. The approach proposed in this paper is to carry out non-intrusive fault detection by externally monitoring of execution time behavior of the application software running on the PPC by using a set of hardware implemented execution timers; and carry out fault handling though mode changes of the application, as shown in Figure 3.



Fault Detection
External Monitoring using hardware implemented execution timers (MC)
- •Synchronize MC and application Tasks at both startup and mode change
- •Detect timing bound violation

Fault Handling
Provide continued service through a predetermined set of application operating modes and monitoring modes
- •Use control flow of tasks to modify timing behavior (switch mode)
- •Switch MC monitoring mode

Fig. 3. Overarching Approach for Fault Detection and Handling

Fault Detection

The timing bounds of each of the application tasks are specified in terms of the worst case execution time (WCET) and the best case execution time (BCET). These bounds are implemented as timers in hardware, and the set of timers associated with the application is referred to as the Monitoring Chip (MC). The MC is implemented on the FPGA along with RavenHaRT as shown in Figure 4.

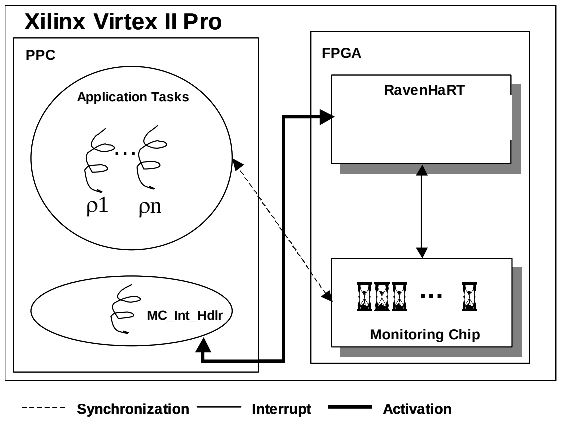


Fig. 4. Fault-Detection Using MC

When a timing bound violation is detected, MC informs RavenHaRT through an interrupt mechanism. The interrupt raised by MC is processed by RavenHaRT, which then activates the MC_Int_Hdlr task in order to switch the application's mode of operation. The timers within the MC are selected based on the application's mode of operation, and the timing bound violated (BCET/WCET). The application tasks on the PPC and the execution timers running in the MC are synchronized by RavenHaRT to ensure that there are no false alarms raised by the MC.

Mode Change

Real and Crespo [14] identify the four requirements for a successful mode change as schedulability, periodicity, promptness and consistency. Each requirement is addressed in the context of the proposed approach:

- *Schedulability* - In the uniprocessor environment provided by the Xilinx Virtex II Pro, at most one task violating its deadline is detected at any given time hence the mode switch is restricted to changing the control flow of a single task. All other tasks continue to operate in the previous mode. Hence the only task whose deadline changes is the aberrant task. The timing behavior of the application tasks are modeled in UPPAAL for the required operational modes and the schedulability is verified prior to system implementation.

- *Periodicity* –The periodicity requirement is satisfied by the RavenHaRT scheduler, which ensures the activation of periodic tasks.

- *Promptness* – The mode change handler receives the identity of the aberrant task, and the bound (BCET/WCET) that it violated. The mode change is carried out based on the priority of the task, and the impact on dependent tasks.

- *Consistency* – The use of protected objects for inter-task communication ensures that shared resources are used consistently.

Each application task follows the same template as shown in Figure 5. The first instruction that the task executes within the loop is a call to the Check_Mode function to read the MODE protected variable present in the SWITCH protected object. This MODE variable determines the control flow of the application task. The different paths through the program converge before the task delays itself or loops. The Change_Mode procedure of the SWITCH protected object is the only way to change the value of MODE, and is accessed by the MC_Int_Hdlr task to issue a mode change instruction. The MC has a state machine, which determines which set of timers to use in the new operating mode based on the current mode of operation and the bound that was violated. The mode switching mechanism is verified by model-checking the operational modes in UPPAAL to ensure that tasks meet their deadlines under degraded operations.

```
task body App_Task_x is
operation_mode : Mode_Type
-- other declarations
begin
 loop
 -- Do something
 Operational_Mode := Switch.Check_Mode;
  case Operational_Mode is
   when   Nominal=> --take one path;
      null;
   when Mode_1 =>-- take another path
      null;
   -- do other paths if needed
   end case;
   --requisite delay statement
 end loop;
end App_Task_x;
```

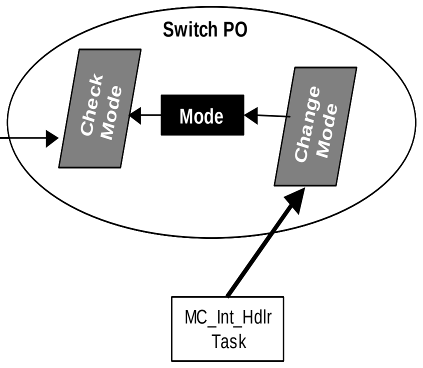


Fig. 5. Fault-Detection Using MC

Switchback capability to nominal mode of operation uses the same functional pieces that operate the switch to the different operating modes. The task violating its bounds continues to run, and the rest of the system is configured to behave as if the faulty task does not exist; i.e., the faulty task is quarantined and the system does not rely on its services. The monitoring chip inspects the timing behavior of the quarantined task. If it runs and communicates with POs nominally then its timing should correspond to the nominal timing. One simple means of quarantining and monitoring task behavior is to modify the timing behavior such that the best case execution time of the quarantined task exceeds the worst case execution time when the task runs nominally, i.e., the MC sets the *BCET'* of the task equal to WCET (where the ' symbol indicates quarantined task). The MC detects the restoration of normal services if *BCET'* is now violated.

Gurkh Generic Navigation System

The Gurkh Generic Navigation System (GGNS) models the core real-time software architecture of a generic guidance and navigation system. The model provides enough functional complexity to be challenging and is small enough for the MC to be synthesized on the FPGA along with the hardware implemented run-time kernel. The GGNS model computes navigation information, such as position, velocity and acceleration, based on two sensors: the Inertial Measurement Unit (IMU) and the Global Positioning System (GPS). GPS data enters the system in the form of messages that are processed to yield Line-Of-Sight (LOS) data, which is fed into a Kalman Filter (KF). The KF estimates present and future navigation information and corrects these estimates according to incoming LOS navigation data. These estimates are fed to a high rate Sequencer task. The Sequencer acts as the central node, gathering all inputs and performs the actual navigation computations. The Sequencer can also request an immediate estimate from the KF if the data is not provided earlier.

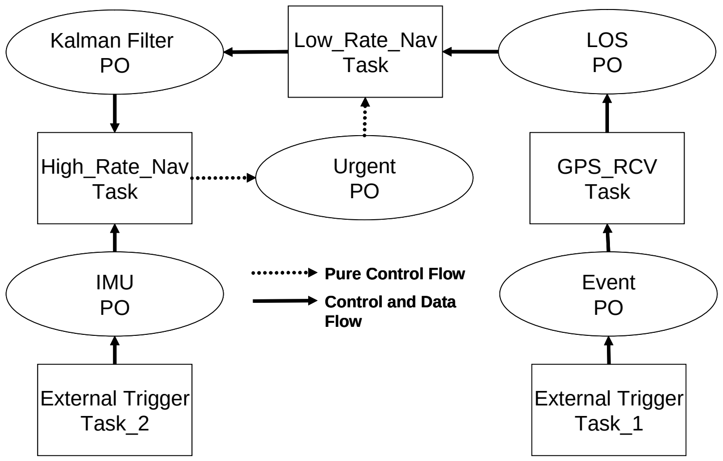


Fig. 6. Task Model of GGNS

GGNS Task Model

The GGNS task model is shown in Figure 6. The system consists of five tasks, three internal to GGNS and two external trigger tasks that simulate input data streams. The High_Rate_Nav task, acts as the overall sequencer, which provides navigation data to the external world. The Low_Rate_Nav task acts as the Kalman filter, carrying out estimation of navigation information by integrating information from both the GPS and IMU. The GPS_RCV task is an event triggered task that gathers LOS data from External_Trigger_Task_1.

The High_Rate_Nav task collects IMU data, KF data, and raises a flag if KF data is not available fast enough. The Low_Rate_Nav task has three activities: collect LOS data, send KF data, and responds to the flag raised by the High_Rate_Nav task by outputting KF data as soon as it becomes runnable. The GPS_Receive task is triggered by the arrival of a GPS message and has two activities: collect the GPS message, and send LOS data.

The data communications between these three main tasks are implemented with protected objects (POs). Three of the POs are the buffers containing LOS data, KF navigation data, and IMU data. The Urgent PO implements the signaling capability that needs to exist between the High_Rate_Nav and the Low_Rate_Nav Tasks. Event PO implements the event triggering capability of the GPS Receive Task. The External_Trigger_Task_1 simulates incoming GPS messages and interacts only with

Event PO. External_Trigger_Task_2 simulates incoming IMU data and interacts only with the IMU data buffer.

Monitoring and Mode Switching

Operational modes are organized hierarchically based on timing behavior violations of the three tasks: High_Rate_Nav, Low_Rate_Nav and GPS_Receive. There are eight possible combinations of task's violating their deadlines, as shown in Figure 7. The hierarchical organization serves to illustrate the different classes of degraded service in decreasing order of the quality of navigation information generated. The modes of operation were determined through interaction with domain experts to ensure that the requisite level of service was maintained. It must be noted that a mode change can only be made along the path from the nominal mode of operation to complete violation of all timing bounds.

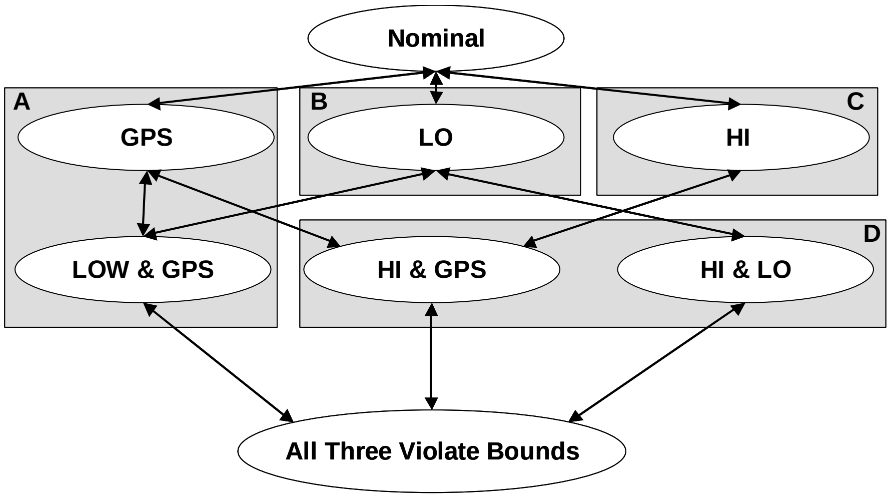


Fig. 7. GGNS Operational Modes

The High_Rate_Nav (HI) task receives input from both the Low_Rate_Nav task (LO), as well as the External_Trigger_Task_2 (which provides inertial measurement data. If the GPS_RCV task (GPS), or the LO violates their timing bounds, HI will not have access to reliable GPS information. The designer may however choose to completely ignore GPS data, switch to Mode A – which is the approach adopted in the implementation. If the system designer works under the assumption that some raw information is more useful than no information, the system bypasses LO temporarily and transitions to Mode B. The system switches to Mode C if HI violates its execution time bound while waiting for IMU data. If both HI and LO violate their deadlines, or both HI and GPS violate their deadlines, then it is essential to ensure that HI can fulfill its essential responsibilities without reliable GPS information, which yields the operating mode D. In the case in which all tasks violate their bounds, the system transitions into safety mode. The summary of the system fault modes is presented in Table 1.

Operating Mode	**High_Rate_Nav**	**Low_Rate_Nav**	**GPS_Receive_Task**
A	Degraded GPS Mode	Holding Error States	Quarantined
B	Degraded LOS Mode	Quarantined	Nominal
C	Basic Mode and Quarantined	Nominal	Nominal
D	Basic IMU Mode Only	Quarantined	Quarantined
E	Survival	Survival	Survival

Table 1. GGNS Modes of Operation

Conclusions

The Gurkh Generic Navigation System is used as a proof-of-concept demonstrator for monitoring the timing behavior of a system with multiple operating modes. The faults are detected based on violation of expected timing behavior, and degraded system performance is guaranteed by modeling system behavior in the presence of faults and formally verifying that the degraded system behavior is deterministic. The current system model and implementation assumes a static set of possible configurations of the system operation in the presence of faults. The operational mode is selected, and the mode transition is predetermined at system implementation time. It must be noted that the approach is currently limited to handling violations of timing behavior that are caused by non-replicable transient faults. This assumption is made to address the limited computational resources available for advanced fault-detection algorithms at the subsystem level. The mode change protocol used is based on the modifying the control flow of tasks that violate their deadlines, and thereby modifying their timing behavior independent of unaffected tasks. The analysis of mode change timing behavior is carried out offline, to ensure schedulability.

An alternative approach is to allow an external system master to determine the subsystem transition mode. This will provide the system the ability to reconfigure itself based on the complete system state (as opposed to the state of just the component such as GGNS). For example, the avionics system may be able to reconfigure position information based on an alternative sensor such as the radar subsystem or the star tracker, in which case the system may choose to transition to the suboptimal operational mode. Work is currently underway to provide reconfiguration support by using dual operating system: RavenHaRT for regular operation and a reconfiguration OS (implemented as an application task), which will determine how the application software running on the PowerPC changes mode, and which monitoring model is used for configuring the MC. Determining the operating mode externally introduces significant challenges in terms of scheduling the mode change, as the subsystem cannot be idle until the mode change request comes through. The system master has to have visibility in terms of affected and unaffected tasks in any given mode, in order to make an informed mode change request.

References

1. Alur, R., D.L. Dill., "Automata for modeling real-time systems", In Proc. of Int. Colloquium on Algorithms, Languages, and Programming, LNCS 443:322-335, 1990
2. Burns A., “The Ravenscar Profile”. *ACM Ada Letters*, XIX, 4, 49–52, Dec 1999
3. Burns, A., “How to Verify a Safe Real-Time System: The Application of Model Checking and Timed Automata to the Production Cell Case Study”, Real-Time Systems, 24, 135-151, 2003, Kluwer Academic Publishers, The Netherlands, 2003
4. Burns, A., B. Dobbing, T. Vardanega, “Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems,” University of York Technical Report YCS-2003-348, 2003
5. de la Puente, J.A., and J. Zamorano, “Execution-Time Clocks and Ravenscar Kernels”, Ada Letters Vol.XXIII, No.4, December 2003
6. Behrmann, G., A. David, and K.G. Larsen, "A Tutorial on UPPAAL", In proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04). LNCS 3185. 2004
7. Harbour M.G., M.A. Rivas, et.al., “Implementing and Using Execution Time Clocks in Ada Hard Real-Time Applications”, Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies, pp 90-101, June 08-12, 1998
8. Hopcroft, J.E. , J.D. Ullman, "Introduction of Automata Theory, Languages, and Computation", Addison Wesley, 2001
9. ISO/IEC Ada 95 Reference Manual, Language and Standard Libraries, Version 6.0
10. Lee, P.A. and Anderson, T., “Fault Tolerance: Principles and Practice (Second Revised Edition)”, Springer-Verlag Wien-New York
11. Liu C.L., J.W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”, Journal of the ACM (JACM), 20(1): 46-61, Jan. 1973
12. Leung J. Y. T. and J. Whitehead, “On the complexity of fixed-priority scheduling of periodic real-time tasks. Performance Evaluation”, 2(4):237--250, Dec 1982.
13. Pettersson, P., and K.G. Larsen, “UPPAAL2k”, Bulletin of the European Association for Theoretical Computer Science, volume 70, pages 40-44, 2000
14. Real J., A. Crespo, “Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal”, Real-Time Systems , 4:161-197, 2004
15. Ram Murthy, C.S., G. Manimaran, “Resource Management in Real-Time Systems and Networks”, The MIT Press, Cambridge, Massachusetts, 2001
16. Silbovitz, A., “RavenHaRT- A Hardware Implementation of a Ravenscar Compliant Kernel”, SM Thesis, Aeronautics and Astronautics, MIT, 2003
17. Torres-Pomales W., “Software Fault-Tolerance: A Tutorial”, NASA Technical Report, NASA-2000-tm210616, 2000.
18. “Virtex-II Pro Platform FPGA Handbook”, v1.0, 2002, www.xilinx.com