A Fault Resilience Tool for Embedded Real-Time Systems

Franklin Lima Santos[1], Flávia Maristela Santos Nascimento[2]

[1] Federal University of Bahia
Department of Electric Engineering
franklin_lima@ieee.org
[2] Federal Institute of Bahia
Department of Electro-Electronic Technologies
flaviamsn@ifba.edu.br

Abstract. Real-time systems have been used in many different areas such as medicine, multimedia and mechatronics. For such systems, it is important to meet both logical and timing requirements, since a malfunction may have undesired consequences. In this paper, we developed a simulation tool in MATLAB® environment to deal with fault-tolerant real-time scheduling under Rate Monotonic scheduling policy, so that errors consequences can be envisioned, before system is put on operation.

Keywords: Real-time systems, fault-tolerance, simulation, MATLAB®.

1 Introduction

Over the past decades computer designers focused their attention on developing what they considered a perfect computer project: computers had to be small, fast and cheap [12]. Indeed, their effort in reaching more performance at low cost and minimum size contributed remarkably for recent technological advances, especially those related to hardware improvements. The remarkable growth of electronic devices and computing systems in our daily activities has been boosting mechatronics, as a subarea of automation due to its ability of integrating electronic components and systems [4, 7]. The main elements of a mechatronic system can be observed in Figure 1.

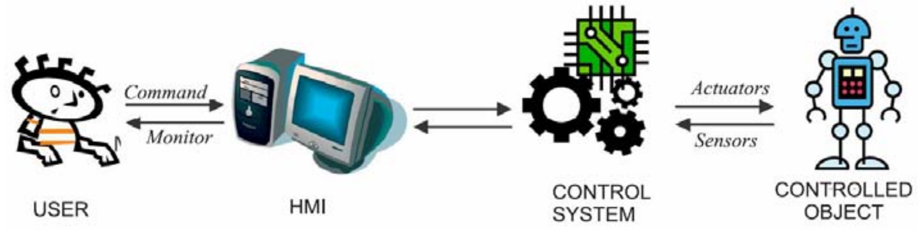


Fig. 1. Components of a mechatronic system [10].

The *user* is the entity responsible for monitoring, super visioning and controlling the *controlled object*, which may be an airplane board control or an industrial plant, for example. *Controlled objects* are usually manipulated through *human-machine interface* (HMI), which is the element responsible for (i) translating control information to the user and (ii) allowing an interface between users and controlled objects. The *control system* is an interactive computer system that enables monitoring and changing the state of a controlled object, which is done through sensors and actuators [8].

The evolution of computer systems also allowed systems designers to focus on modeling, designing and implementation aspects of such systems aiming at developing applications with differentiated performance skills. At the same time, miniaturization of electronic components allowed computers to evolve from simply terminals to host control systems. For some of these applications correctness were not only associated with logical, but also with timing requirements. Indeed, systems in which correctness is associated not only with producing logically correct results, but also with the time at which such results are produced (timeliness) are known as *real-time systems* [2,5].

Real-time systems are present in many different areas such as medicine, avionics, multimedia and mechatronics [13]. For some of them, when timing requirements are not accomplished, the system may not achieve the expected level of Quality of Service (QoS). This is what happens, for example, in a video transmission (multimedia application). In worst cases, missing timing requirements may have undesired consequences as for example, if we consider an automobile ABS control, in which human life may be at risk [3]. This evidenced that different applications may have different criticality levels. Indeed, for "soft" real-time systems missing deadlines may not have more serious consequences, while for "hard" real-time systems missing deadlines may cause injuries for human beings and/or environment [2, 9, 11].

Since temporal requirements play an important role for real-time systems, it is crucial to have means of guaranteeing such requirements. In fact, both *scheduling policies* and *schedulability analysis* are responsible for ensuring timeliness for such applications. To do so, system tasks are ordered according to a specific scheduling policy and a subsequent schedulability analysis is performed to assess timeliness of each task. We detail such aspects in Section 2.

Ensuring reliability is an important goal to be achieved for real-time systems. However, in terms of computational applications, the only certainty we have is that all of them may potentially fail [1]. In fact, system correctness relies on its dependability, a concept which discussed in Section 3. Also, since faults cannot be avoided and are difficult to predict [9, 11], taking such events into consideration is almost an obligation, if someone needs to guarantee QoS for real-time applications or even avoid more serious consequences.

In this paper we investigate the impact of errors in real-time applications considering a specific scheduling policy. To do so, we defined a simulation environment, presented in Section 4 and developed a simulation tool, detailed in Section 5, which aims at measuring fault resilience for a particular set of real-time systems. Last, in Section 6, some conclusions and future works are drawn.

2 Real-Time System Overview

A real-time system is a computer system in which both timing and logical requirements must be respected. Thus, the correct behavior of such a system depends not only on the integrity of produced logical results (also known as "correctness"), but also on the time at which they are produced ("timeliness") [2]. Examples of real-time systems include current control laboratory experiments, vehicle control, nuclear plants and flight control systems [13].

Usually, real-time systems are structured as a set of *n* periodic tasks $\Gamma = \{\tau_1, \tau_2, \ldots \tau_n\}$. A given task τ_i represents a function, routine (or subroutine) or any code snippet. Each task τ_i has attributes such as an execution cost C_i, a deadline D_i, an activation period T_i and a recovery execution cost $\overline{C_i}$. Thus, a periodic task can be described as an ordered tuple $\tau_i = (C_i, T_i, D_i, \overline{C_i})$.

Tasks are executed in a specific order called execution scale. Such a scale is defined based on some heuristics, known as *scheduling policy*. Several scheduling policies have been addressed in literature and most of them are priority oriented [3, 9, 10], which means that tasks are ordered according to its priority.

A well-known priority oriented scheduling policy is Rate Monotonic (RM), according to which tasks with shortest periods have higher priority. Clearly, this is a fixed-priority policy, since tasks period are defined offline and do not change during system execution.

After a scheduling policy is chosen for a given task set Γ is it important to assess if any task τ_i □ Γ may miss its deadline. To do so, we perform some tests, also known as schedulability analysis, which aims at determining if a given task set is *feasible*. In other words, such tests determine if any task τ_i □ Γ misses its deadline. Clearly, schedulability analysis is strongly linked with the chosen scheduling policy. In this paper we address the analysis based on Processor Utilization Analysis, which is discussed in Section 2.1.

2.1 Processor Utilization Analysis

According to this approach, the schedulability of a given task set is assessed based on processor use. Indeed, processor utilization *U*, for a given task set Γ composed of *n* a periodic and independent real-time task is given by:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{1}$$

Regarding Rate Monotonic, if we assume a periodic task set Γ in which tasks period are equal to their deadlines, we state that Γ is schedulable if:

$$U \leq n(\sqrt[n]{2} - 1) \tag{2}$$

For RM, Processor Utilization Analysis is a sufficient schedulability test, which means that it is not able to ensure schedulability for all task sets. In fact, it has been proven that [6] if:

$$U \leq \ln 2 \quad (3)$$

The task set is schedulable with RM. Otherwise the analysis does not guarantee schedulability. Also, Rate Monotonic is considered an optimal algorithm for systems in which tasks period are equal to their deadlines ($T_i = D_i$) [6].

3 Fault-Tolerant Real-Time Systems

Faults are random events that cannot be predicted or avoided. Actually, the only certainty we have is that all computational applications potentially fail [1]. Indeed, a fault may be caused by several different events, as for example cosmic radiation, hardware fatigue or malfunctioning, specification and/or implementation aspects.

A system is said to *fail* when there is a transition from an expected correct behaviour to an incorrect and unexpected behaviour. In other words, a fail represents a deviation from specification. The *error* is the state that leads the system to fail and *faults* are the causes of an error, which may be physical or algorithmic [1]. Indeed, applications must provide confidence in the expected operations, a concept usually addressed as *dependability*, which is related to some attributes such as availability, reliability, safety and maintainability [1, 14].

In terms of real-time system there is a concern about fault tolerance aspects, since a fault may affect the system schedulability, or in other words, may prevent tasks to meet their deadlines. For this reason, faults are considered as a threat to dependability. Thus, techniques must be implemented to deal adequately with faults, so that applications keep their correctness even in the presence of such events [1, 14].

Faults are more commonly classified based on the persistence criterion, according to which they can be transient, intermittent or permanent. *Transient* faults occur only for a given time and then disappear. An example could be electromagnetic interference. When a transient fault occurs repeatedly it is called *intermittent*, as for example a loose contact on a connector. Both transient and intermittent faults are difficult to diagnose. Last, *permanent* fault is one that continues to exist until the faulty component is repaired, as for example a lack of connectivity between two nodes in a network [10, 14].

In this paper we investigate the effects of transient faults focusing on techniques that can be used to deal with such faults, which are based on temporal redundancy. This consists of repeating the computation in time, or in terms of scheduling can be understood as re-executing a task (see Figure 2) or executing an alternative task (see Figure 3) until the system is put on a safe state [9,11].

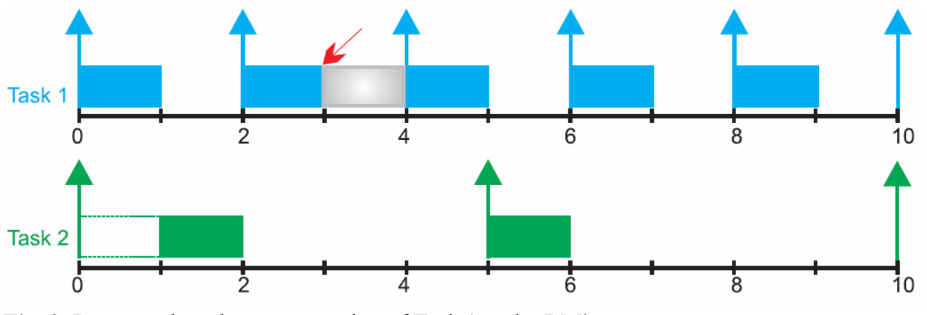


Fig. 2. Recovery based on re-execution of Task 1 under RM}

Figures 2 and 3 presents a periodic task set being scheduled, where $\Gamma = \{\tau_1 = (1,2), \tau_2 = (1,5)\}$. Observe that in Figure 2 an error occurred at $t = 3$ (red arrow), which affect Task 1. The faulty task re-executed immediately since there were no other higher priority task to execute. On the other hand, in Figure 3, an error affected Task 2 at $t = 6$ (red arrow), but it only could recover at time $t = 7$, since Task 1 was already released for execution and has a higher priority than Task 2.

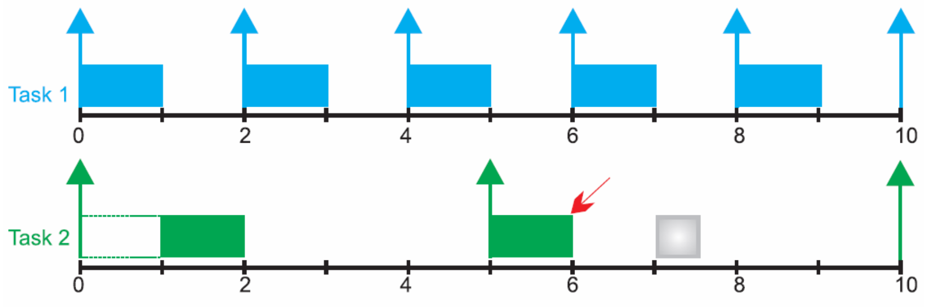


Fig. 3. Recovery based on the execution of an alternative version of Task 2 under RM.

In the following sections we present the developed tool which focus on measuring the resilience of hard real-time systems scheduled according to RM scheduling policy.

4 Simulation Environment

4.1 System Model

The assumed system model considers a task set Γ composed of n independent and periodic real-time tasks $\Gamma = \{\tau_1, \tau_2, \ldots \tau_n\}$. Each task τ_i is represented by a tuple $\tau_i = (C_i, T_i)$ where C_i is the constant worst-case execution time (wcet) of each task and T_i

is the activation period. Also, we assume that the deadline for each task is the same as its period ($T_i = D_i$).

Tasks are scheduled according to Rate Monotonic, since this algorithm deals with fixed priority tasks and is widely used for embedded critical applications. Also, schedulability analysis is performed with Processor Utilization Analysis.

4.2 Fault Model

Assuming a specific fault model is a difficult task since faults are random and cannot be predicted. We consider that the system is subject to multiple transient faults which can occur at any time instant.

Also, we represent the fault resilience of a given system through the maximum number of errors the system can handle and keep its correct behavior. To do so, we use a random function in MATLAB® to generate the number of errors that will affect each system. Also, the time instant in which errors occur is also determined through a random procedure.

We discard errors that occur at time instants in which no task is executing, since such errors will not affect system behavior. We assume that fault detection occurs implicitly, at the end of each task execution, since the focus of the work is not the detection procedure, but system behavior after recovery strategies.

4.3 Recovery Model

The recovery model describes the strategy used to put the system in a safe state. Indeed, we consider two possible actions: (i) faulty task re-execution or (ii) execution of an alternative task. Both strategies are defined offline, before running the system, and are performed in idle time instants available in execution scale.

5 Simulation Tool

The general overview of the developed tool can be seen in Figure 4. The tool was developed in MATLAB®, due to its versatility on numerical analysis, encapsulated functions and graphics.

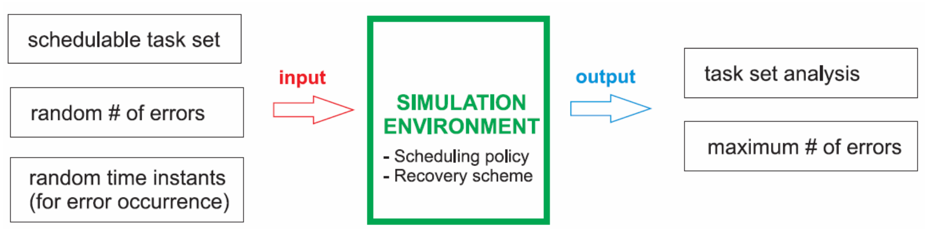
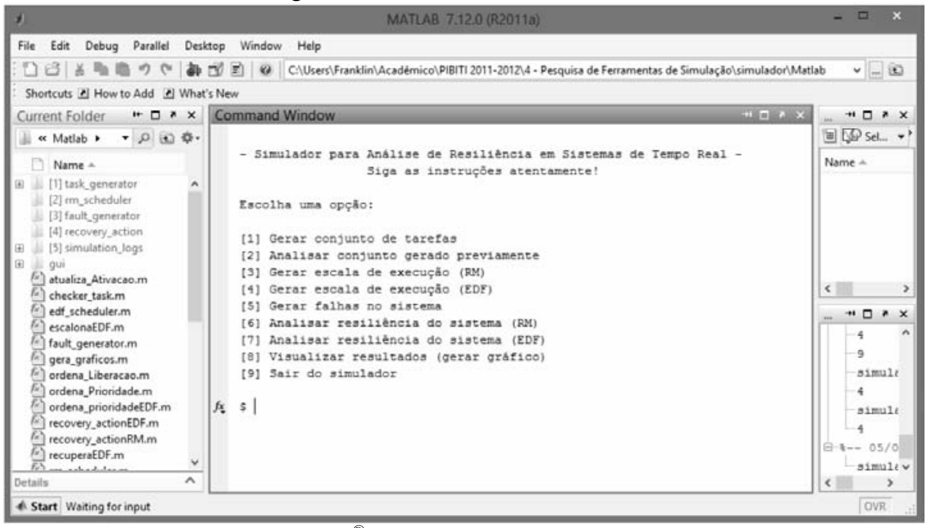


Fig. 4. Framework of Simulation Environment

The first step to use this tool is to input a *schedulable task set*. In case the user has no previously generated task set, it is possible to generate a random one inside developed environment. To so, the user only has to choose the number of tasks to be generated. In case the tool generates the task set, it also tests if it is schedulable, through processor utilization analysis.

After, the user has to generate the *number of errors* that will affect the task set. As mentioned before, such a number is randomly generated by the tool. The user only defines a lower and upper bound, which will represent the interval in which the number of errors will be in. Based on the number of errors, the tool generates *random time instants* in which errors will occur. The screen of MATLAB® running the simulator can be seen in Figure 5.



```
- Simulador para Análise de Resiliência em Sistemas de Tempo Real -
                  Siga as instruções atentamente!

Escolha uma opção:

[1] Gerar conjunto de tarefas
[2] Analisar conjunto gerado previamente
[3] Gerar escala de execução (RM)
[4] Gerar escala de execução (EDF)
[5] Gerar falhas no sistema
[6] Analisar resiliência do sistema (RM)
[7] Analisar resiliência do sistema (EDF)
[8] Visualizar resultados (gerar gráfico)
[9] Sair do simulador

$ |
```

Fig. 5. Screen shot of MATLAB® running simulator.

The *simulation environment* will generate an execution scale, which takes into consideration Rate Monotonic, as scheduling policy, the defined recovery scheme (re-execution or alternative task code) and the time instants in which errors occur. Based on those values, the system resilience is defined and results can be graphically checked.

Briefly, the simulator executes the following steps, given the inputs described in Figure 4:

- Identify tasks affected by errors;
- Identify idle time after each faulty task, which can be used for recovery;
- Verify the possibility of re-executing the faulty task or executing an alternative code, respecting tasks priority (including the simultaneous verification of space for recovery and maximum execution time);
- Graphically analyze the resilience of the system, through graphically generated execution scale.
- Inform the number of errors and time instant which makes the system unschedulable.

To make things clear let us consider the following example:

Example 5.1. Assume a task set $\Gamma = \{\tau_1, \tau_2\}$ composed of two independent and periodic tasks where $C = (3, 3)$ and $D = T = (8, 12)$. Tasks are scheduled according to RM and in case of faults, tasks are re-executed. In other words, $\overline{C_i} = C_i$.

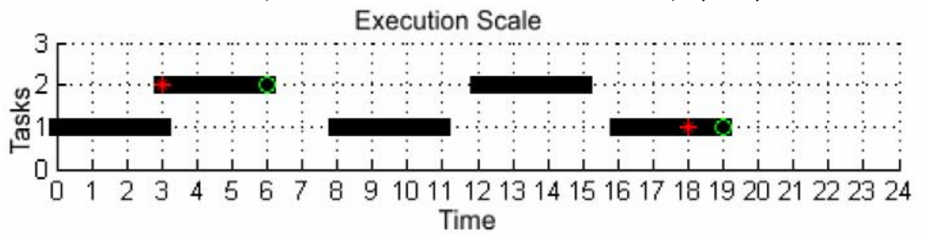


Fig. 6. Execution Scale for Example 5.1.

The system is simulated during the hyperperiod $h = lcm(8, 12) = 24$[1] to ensure that all system execution will be considered.

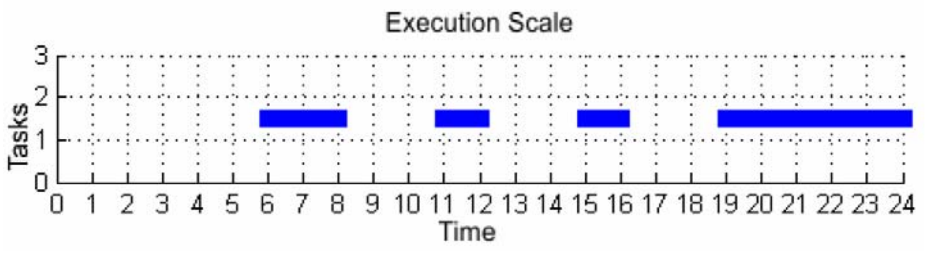


Fig. 7. Idle processor time for Example 5.1, graphically represented in tool.

The first chart presented in Figure 6 presents the execution scale for the given task set. The random number of faults that this task set is subject to is $n_f = 2$ and the random time instants in which they occur was $t_f = (3, 18)$. This is shown by a red mark in the chart. Detected errors are indicated by green circles.

Figure 7 evidences the idle processor time, which are represented in blue. Finally, Figure 8 presents the fault-tolerant real-time schedule.

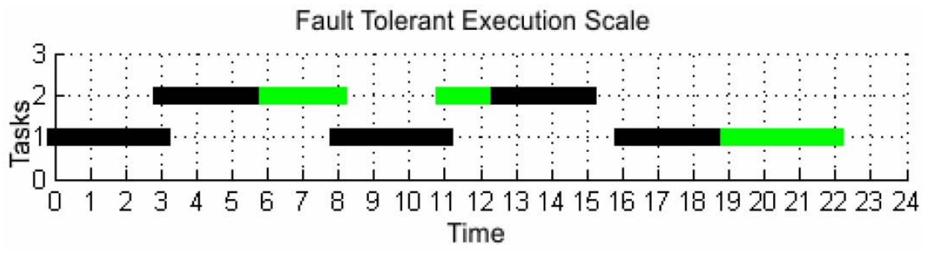


Fig. 8. Fault Tolerant Scheduling for Example 5.1 assuming errors at $t_f = (3, 18)$.

[1] *lcm(x, y)* is the function which calculates the least common multiple of input parameters ,in this case, x and y. Usually systems are simulated during the hyperperiod, since it contains all system behavior.

It is important to mention that depending on the time instant that errors occur, the system may not be schedulable, even if it is subject to the same number of errors. Observe Figure 9, which presents the same task set described in Example 5.1 subject to two errors that happens at $t_f = (2, 3)$.

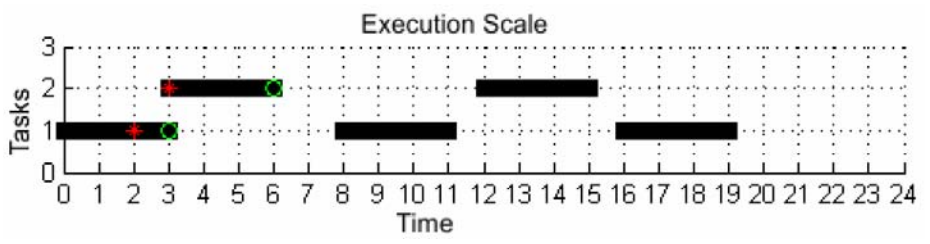


Fig. 9. Execution Scale for Example 5.1 assuming errors at $t_f = (2, 3)$.

Observe that in this case, recovery of both faulty tasks is not possible, since the available idle time (same as presented in Figure 7) is not enough for recovering tasks τ_1 and τ_2. before their respective deadlines. The graphic presented by simulator is according to Figure 10, confirming that the fault-tolerant scheduling is not feasible.

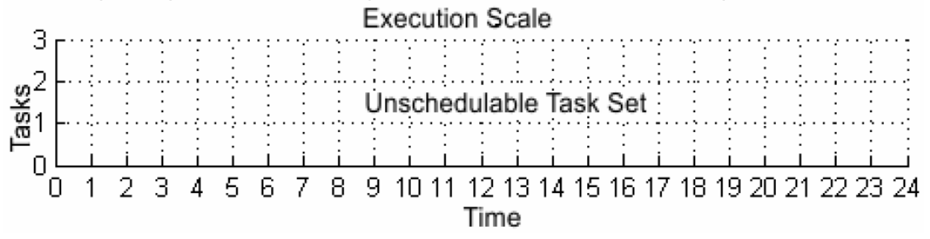


Fig. 10. Fault Tolerant Scheduling for Example 5.1 assuming errors at $t_f = (2, 3)$.

6 Conclusions and Future Work

Real-time systems have been used in a wide range area, as for example to control industrial processes. For most of these applications, missing timing requirements imply in a loss of Quality of Service or in worst cases may cause social, economic and/or environmental injuries. In this context, it is extremely necessary to deal with unpredictable and random events, such as faults, so that they interfere minimally in systems operation. In this paper we developed a simulation tool in MATLAB® environment to deal with fault-tolerant real-time scheduling, so that errors consequences can be envisioned, before system is put on operation.

One of our goals is to have an approximation between theoretical and practical models. This will enable more detailed studies and previous use of simulations before the applications are put into production. As future work we aim at simulating more robust systems, to evaluate better our preliminary results. Also, we focus on extending scheduling policies, so that EDF [6] is also considered.

7 Thanks

The authors would like to thank the Federal Institute of Bahia (IFBA) for all support and National Council for Scientific and Technological Development (CNPq) for funding this work.

References

1. Algirdas Avizienis, Jean-Claude Laprie, Carl Landwehr, and Brian Randell. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, 1(1):11–33, 2004.
2. Jean-Marie Farines, Joni da Silva Fraga, and Rômulo Silva de Oliveira. Sistemas de Tempo Real. Departamento de Automa¸c˜ao e Sistemas - Universidade Federal de Santa Catarina, Florianópolis, Santa Catarina, 2000.
3. Sunondo Ghosh, Rami Melhem, and Daniel Moss´e. Fault-Tolerant Scheduling on a Hard Real-Time Multiprocessor System. In Proccedings of Eighth International Symposium on Parallel Processing, pages 775 – 782, April 1994.
4. Rolf Isermann. Modeling and design methodology for mechatronic systems. IEEE/ASME Transactions on Mechatronics, 1(1):16–28, 1996.
5. Li Jie, Guo Ruifeng, and Shao Zhixiang. The Research of Scheduling Algorithms in Real-Time System. In International Conference on Computer and Communication Technologies in Agriculture Engineering (CCTAE'10), volume 1, pages 333 – 336, June 2010.
6. C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM, 20(1):46–61, 1973.
7. Ren C. Luo. Sensors and actuators for intelligent mechatronic systems. In 27th Annual Conference of the IEEE on Industrial Electronics Society, 2001. IECON'01, volume 3, pages 2062–2065, 2001.
8. Paulo Eigi Miyagi and Emilia Villani. Mecatrônica como solução de automação. In Revista Ciências Exatas. Universidade de Taubaté, 2004.
9. Flávia Maristela Nascimento, George Lima, and Verônica Cadena Lima. Deriving a fault resilience metric for real-time systems. In Workshop de Testes e Tolerância a Falhas, August 2009.
10. Flávia Maristela Santos Nascimento. A Simulation-Based Fault Resilience Analysis for Real-Time Systems, 2009.
11. George Lima, Flávia Maristela Santos Nascimento, Verônica Lima. Fault resilience analysis for real-time systems. In Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems, pages 35 – 38, Brussels, Belgium, October 2010.
12. Mircea R. Stan and Kevin Skadron. Power-Aware Computing. Computer, 36:35 – 38, December 2003.
13. John Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. Computer, 21(10):10–19, 1988.
14. Andrew S. Tanenbaum and Maarten van Steen. Distributed Systems: Principles and Paradigms. Prentice Hall, 2006.