Deriving a Fault Resilience Metric for Real-Time Systems [*]

Flávia Maristela Nascimento[1], **George Lima**[2], **Verônica Cadena Lima**[3]

[1]Department of Technology in Electro-electronics
Federal Institute of Education, Science and Technology of Bahia (IFBA)
41.301-015 – Salvador – BA – Brazil

[2]Department of Computer Science – Federal University of Bahia (UFBA)
41.170-110 – Salvador – BA – Brazil

[3]Department of Statistics – Federal University of Bahia (UFBA)
41.170-110 – Salvador – BA – Brazil

flaviamsn@ifba.edu.br, {gmlima,cadena}@ufba.br

Abstract. *Most real-time systems are required to comply with strict time and logical requirements even in the presence of faults. Although scheduling policies and schedulability analyses have been extended to deal with fault tolerance, not much attention has been given to measuring the fault resilience of such systems. Usually, worst-case error patterns are artificially assumed and system correctness is checked. However, such patterns do not represent the capacity of the system to tolerate faults, nor consider the overall system behavior in the presence of faults. In this paper we define a fault resilience metric and present a simulation-based analysis. Then we show how simulation results can be statistically analyzed.*

1. Introduction

1.1. Motivation

Real-time systems are characterized by their need to meet both logical and timing requirements. On the assumption that any system potentially fails [Avizienis et al. 2004] it is necessary to ensure that such requirements will be met even in the presence of faults [Burns and Wellings 2001]. For example, an error or deadline miss in a flight control system may incur in loss of human life.

In order to guarantee that deadlines are met in a real-time system, all tasks have their execution ordered according to some heuristics, namely scheduling policy, taking into consideration their timing requirements. Given a scheduling policy, timing correctness can be checked by means of schedulability analysis. If fault-tolerant techniques are considered both scheduling policy and schedulability analysis have to be adapted to take their execution into account. In general, schedulability analysis is built up on the assumption that the system behavior is known in the worst-case and this has also been true when fault tolerance aspects are considered.

Despite the difficulty in incorporating fault tolerance mechanisms into schedulability analysis, several scheduling techniques have been developed. Most of them are

[*]The second author is supported by FAPESB under grant 7320/2007.

based on time redundancy, which can be achieved by providing error detection and recovery. Models such as recovery blocks or exception handlers [Burns et al. 1996] are particularly useful for real-time uniprocessor systems. In both techniques, upon the detection of an error, a recovery task is scheduled to provide the actions necessary to keep the system correctness. This approach has been extensively used since it is particularly suitable for dealing with transient faults, which has been pointed out as the most frequent one [Ghosh et al. 1998] or permanent software faults.

The usual approach to taking the effects of recovery tasks into schedulability analysis is to artificially assume a given worst-case pattern for error occurrences. For example, some approaches consider that errors take place periodically in worst case. The recovery scheme is based on re-executing of the faulty task or executing an alternative version of it [Lima and Burns 2003, Burns et al. 1996]. Other authors fix a maximum number of errors per system task and recovery is carried out based on the execution of an alternative task [Aydin 2007, Liberato et al. 2000].

Although such approaches are important, since they allow to consider error occurrences in real-time systems, they assume a worst-case error pattern, which may not reflect the real system capacity to tolerate faults. Indeed, considering error patterns as fault resilience metrics may not be suitable due to two main reasons: (a) the assumed error patterns are usually linked to specific system and/or fault models and (b) they do not take the overall system behavior into account, focusing on worst-case scenarios. Thus, deciding which technique best suits an application may not be straightforward. Indeed, applications that do not share the system or fault models cannot be compared straightaway. This means that if one is deciding to implement a system, he/she might not be able to choose the best approach from the fault resilience view point because they are not comparable.

This paper presents a fault resilience metric derived based on some desirable requirements and assumptions. Such a metric is independent of the assumed system model and/or error pattern and can be used to subsidize the system designer decisions when choosing the fault-tolerant mechanisms that best suit their systems. We also present a means of deriving the defined metric using simulation. Statistical analysis is then applied so that the overall system behavior can be derived. Experimental results indicate the differences of two well known scheduling policies from the viewpoint of fault resilience, demonstrating that our approach is reasonably independently of the system model.

The remainder of this paper is organized as follows. Some related work, the assumed fault model and notation used throughout this paper are detailed in Section 2. Section 3 outlines some fault resilience metric requirements and assumptions. Section 4 presents the simulation environment built to compute the proposed metric. The derived metric is used to compare two different scheduling approaches for real-time systems and these simulation results are presented in Section 5. Our final comments are given in Section 6.

2. Background and Related Work

2.1. Background

Real-time systems are usually structured as a set of tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$, where each task represents an execution unit. A task may be activated several times during the system execution. Each task activation is usually called a *job*. Some important task attributes are period, relative deadline, worst-case execution time and recovery execution time, which can be described by $\mathbf{T} = (T_1, \ldots, T_n)$, $\mathbf{D} = (D_1, \ldots, D_n)$, $\mathbf{C} = (C_1, \ldots, C_n)$ and $\bar{\mathbf{C}} = (\bar{C}_1, \ldots, \bar{C}_n)$, respectively.

Meeting all task deadlines is an important requirement for hard real-time systems. Indeed, when such a requirement can be met the system is said schedulable. In order to guarantee timeliness requirements for a given system, all tasks in Γ have to be ordered according to some scheduling policy. The most popular scheduling policies are priority oriented. Two well-known are *Rate Monotonic* (RM) and *Earliest Deadline First* (EDF) [Liu and Layland 1973]. The former is a fixed-priority scheduling algorithm according to which task priorities are assigned in inverse order of their periods. The later is a dynamic policy and assigns priority to jobs so that the most urgent ones gets the highest priority.

Given a task set Γ and scheduling policy the role of the schedulability analysis is to check whether tasks meet their deadlines. For example, it is well known that under RM a task set is schedulable if the relation in Equation (1) holds.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \tag{1}$$

For EDF, schedulability is ensured if and only if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2}$$

Usually, the system schedulability is assessed in fault-free scenarios, although these kinds of equations have been extended to take the effects of errors into account [Pandya and Malek 1998, Han et al. 2003]. For example, considering a task set scheduled by RM in which a single error occur during the hyperperiod, which is the least common multiple of task periods, Equation (1) could be easily adapted to

$$\sum_{i=1}^{n} \frac{C_i + \bar{C}_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$$

Assuming that a task is released at time t, its absolute deadline is given by $t + D_i$. In this work, we consider that the relative deadline D_i of each task $\tau_i \in \Gamma$ is not greater than its period T_i. Also, tasks are assumed to be independent of each other and their worst-case execution time (C_i) are known and not greater than $\min(T_i, D_i)$. Fault tolerance is provided by executing recovery actions upon error detection. These actions actually represent the execution of any extra code. A recovery action associated to a given error in τ_i can be the re-execution of τ_i or the execution of an alternative task. If errors are detected during the recovery of τ_i, other recovery actions can be released.

It is interesting to note that this model is in line with most fault tolerance techniques based on temporal redundancy such as recovery blocks or exception handlers [Burns et al. 1996], which have been widely applied to real-time systems [Lima and Burns 2003, Liberato et al. 2000] and can be implemented at the task level. More severe types of errors require spatial redundancy, usually implemented with a distributed/parallel architecture, which is beyond the scope of this work.

2.2. Related Work

In order to consider the possibility of errors in real-time systems some worst-case error pattern is assumed. Then, timeliness assessment is considered under such circumstances. For example, in some approaches errors are assumed to occur once in the hyperperiod [Han et al. 2003, Pandya and Malek 1998], which is defined as the least common multiple of task periods. For others, the minimum time between errors are assumed to be known [Ghosh et al. 1998, Lima and Burns 2003, Burns et al. 1996]. All these assumptions are used to make it possible to incorporate the effects due to errors into the analysis. Nonetheless, they are not suitable to represent the fault resilience of the system. Indeed, the system may cope with more than what was assumed since only worst-case scenarios were considered for schedulability analysis purposes. Some other approaches fix a maximum number of errors per system task [Liberato et al. 2000, Aydin 2007]. Also, deterministic assumption is not in line with the random nature of errors [Pereira et al. 2004].

Fault resilience has been analyzed by some authors via checking whether what has been assumed in the worst case is violated. For example, considering a Poisson distribution for error occurrence, an upper bound on the probability that error take within less than what is assumed has been derived [Burns et al. 1999]. In other approaches, the assumed maximum number of errors was considered to be a function of both a probabilistic fault model and a desired probability threshold [Burns et al. 2003, Broster and Burns 2004]. Nonetheless, since worst-case scenarios hardly occur, these approaches may also be pessimistic. Also, the assumed error patterns are usually strongly linked with the system scheduling model. Thus, it makes it difficult or even impossible to compare different systems from the fault resilience perspective. For example, consider a system that can deal with one error occurrence per hour and other that can cope with N errors within its hyperperiod. Which one is better? Answering this question may not be possible due to the reasons mentioned above.

Unlike these results, we propose a more general metric which is not specific for a given error pattern or system model. Indeed, it aims at measuring the system resilience for different systems and fault models. Although we do not assume a particular scheduling policy, we restrict ourselves to those policies whose scheduled jobs have fixed priorities, such as Rate Monotonic and Earliest Deadline First. A simulation environment was developed to simulate the system during specific time windows and compute, for each of them, the fault resilience metric. Although there are other simulation-based scheduling approaches [Wall et al. 2003, Huselius et al. 2007], fault tolerance aspects have not been considered.

3. On the Fault Resilience Metric

In order to give some intuition on the need for a fault resilience metric, consider the following example.

Example 3.1. *Let Γ be a task set composed of two periodic tasks $\Gamma = \{\tau_1, \tau_2\}$. Assume $\mathbf{T} = (2, 5)$, $\mathbf{C} = (1, 1)$, $\mathbf{D} = \mathbf{T}$ and $\bar{\mathbf{C}} = (1, 1)$. Also, consider that Rate Monotonic is used to schedule the tasks and so τ_1 is the highest priority task. Figure 1 shows the schedule for this task set, considering that no error occurs, for intuition means. Up arrows mean tasks release time.*

The hyperperiod for this task set is given by $h = \mathrm{lcm}(T_1, T_2) = 10$. Notice that τ_1 has five *jobs* within h, which are released at times $0, 2, 4, 6$ and 8, while τ_2 has two jobs

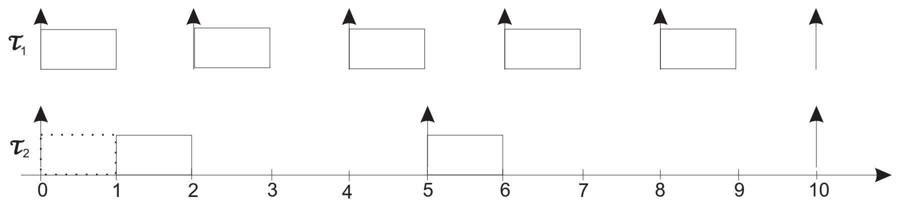


Figure 1. Illustration of a RM schedule for the task set described in Example 3.1

released at times 0 and 5. Because of its priority, the first job of τ_2 begins to execute at time 1. The dotted line indicates that τ_2 awaits until higher priority jobs finish executing.

Assume that recovery is based on the re-execution of the faulty jobs. Figure 2 shows the schedule for the task set shown in Example 3.1. Notice that after the moment the error takes place, at time 2, 6 and 8, the recovery job (shown in gray) executes to keep system correctness. Also, observe that task τ_2 can deal with different number of errors within 5 time units. In the time interval $[0, 5)$ the job of τ_2 can deal with only a single error, while from time 5 to time 10 such a job can timely recover from at most two errors and still meet its deadline.

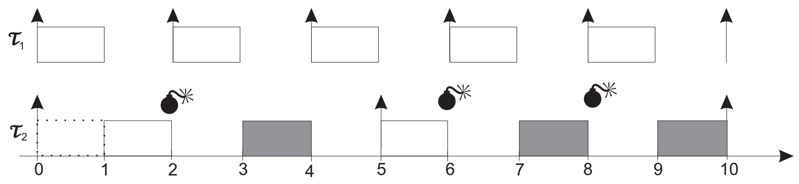


Figure 2. Illustration of τ_2 in Example 3.1 subject to errors within h

The schedules shown in Figure 2 give the intuition that the fault resilience metric must reflect the number of faults the system can tolerate. Thus, assuming that one wishes to analyze the behavior of a given system Γ when it is subject to faults, the analysis method must use some fault resilience metrics. We understand that such metrics must take the following assumption into consideration.

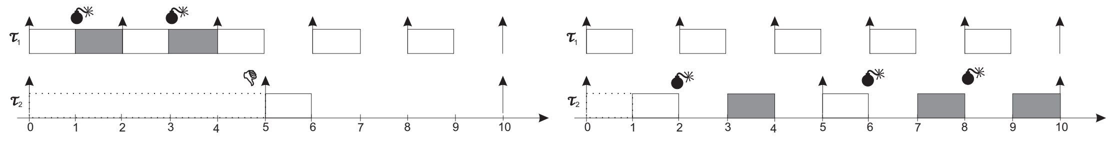
Assumption 3.1. *The fault resilience of a system is proportional to the number of errors it can deal with.*

Indeed, fault resilience metrics must reflect the system ability to survive, or keep its correct behavior, after error occurrences. Most authors are aware of that [Lima and Burns 2003, Pandya and Malek 1998, Liberato et al. 2000, Ghosh et al. 1995], despite assuming specific error patterns. Also, from designers' view point it is important to determine how many errors can occur before a system failure.

In several situations, the number of error occurrences accounted for when analyzing a system must be a function of time. The intuition is that the expected number of errors increases with time assuming that they are not co-related. Although some authors do not consider such assumption [Pandya and Malek 1998, Han et al. 2003], some fault models in real-time systems that are in line with this observation can be mentioned, as for example Poisson distribution [Burns et al. 2003] and minimum time between errors [Burns et al. 1996, Burns et al. 1999]. Based on that, we assume the following:

Assumption 3.2. *The expected number of error occurrences increases with time.*

Since the system we are considering is composed of n tasks and each of them may have a different level of criticality, the fault resilience metric must be determined for each individual task so that system designers can deal with the peculiarities of each of them. Figure 3 shows the maximum number of errors that two different tasks from the same system can cope with.



(a) Maximum number of errors for $\tau_1 \in \Gamma$ (b) Maximum number of errors for $\tau_2 \in \Gamma$

Figure 3. Illustration of two different tasks of Γ subject to errors within h

Considering Example 3.1, τ_1 can timely recover from at most 1 errors within 2 time units, while τ_2 can deal with one or two errors within 5 time units. This is shown in Figures 3(a) and 3(b), respectively. Note that in the figure errors take place at the end of the job execution, the worst-case scenario. Thus, as tasks may have different criticality levels, it is interesting to consider the following requirement:

Requirement 3.1. *Fault resilience must be given for individual tasks of the analyzed system.*

Further, the resilience of a given task $\tau_i \in \Gamma$ depends on how its jobs behave when errors take place. For example, two jobs of τ_i might be capable of tolerating different number of error occurrences during their executions, due to the different interferences they suffer. Observe in Figure 3(b) that the first instance of τ_2 can deal with only a single error. Since τ_1 interferes in its execution, there is not enough time to recover the system from two errors. On the other hand, the second job of τ_2 can timely recover from two errors, since the interference it suffers from τ_1 is smaller, considering that this job is not affected by errors. Indeed, scheduling decisions or the set of interfering jobs may not be the same for all jobs of τ_i. In order to capture the behavior of τ_i as a whole, therefore, different jobs of the analyzed task must be taken into consideration. This motivates the following requirement:

Requirement 3.2. *The fault resilience of a task must account for the overall behavior of its jobs.*

Based on the assumptions and requirements stated above, we give the following definition of fault resilience metric:

Definition 3.1. *The fault resilience of a job is measured as the minimum number of errors that make it miss its deadline divided by its relative deadline. The fault resilience distribution of a task is given by the fault resilience of its jobs.*

According to the above definition, error occurrences are considered for each task in a per-job basis, which is in line with Requirements 3.1 and 3.2. Also, the time windows in which jobs execute (relative deadlines) are taken into consideration. Indeed, the longer the execution of a job the more likely errors occur, which complies with Assumption 3.2. Finally, observe that Assumption 3.1 is also considered.

Obviously, different metrics can be given. Some assumptions/requirements may not be suitable for all systems while new assumptions/requirements may be needed for others. For example, if Requirement 3.2 is not needed, usual analysis based on worst-case scenarios may suffice. Further, if Assumption 3.2 is removed, counting the minimum number of errors per task in worst-case scenarios is enough [Lima and Burns 2005]. In any case, we stress here that we use a metric which is in line with Definition 3.1 as a means of fault resilience assessment.

Motivated by the above requirements and assumptions, we derive a fault resilience analysis based on simulation, as will be explained in the following section.

4. Simulation Environment

The simulation scheme illustrated in Figure 4 is based on modeling the analyzed system with two main components: the *scheduler* and the *error generator*. The former follows a given scheduling policy while the goal of the latter is to generate errors so that jobs deadlines are missed. No particular error pattern is assumed. A role of the error generator is to derive the worst-case error pattern for each simulation. These two components are named *simulation engine*.

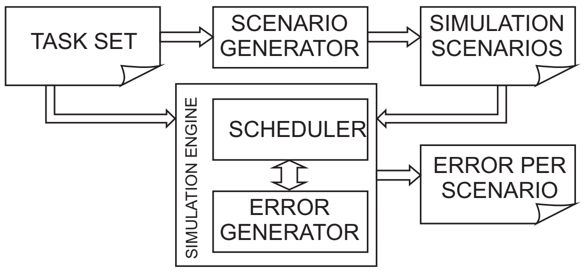


Figure 4. Simulation Environment

The general idea of the simulation-based analysis is to generate possible tuples of tasks release time, which we call *simulation scenarios* and for each tuple compute the minimum number of errors that make a specific job miss its deadline. Then, considering this number of errors per scenario and the simulation time interval, the fault resilience metric is computed. Finally this simulation data can be statistically analyzed to infer the system behavior.

Unlike existing simulation-based analysis, the simulation environment in Figure 4 does not need to simulate the whole system execution (e.g. system hyperperiod), which might be too time consuming in the general case. Instead, it uses simulation scenarios to define the simulation time interval. Formally, simulation scenarios are defined as follows:
Definition 4.1. *Tuple* $\mathbf{S} = (S_1, \ldots, S_n)$ *is a simulation scenario of a periodic task set* $\Gamma = \{\tau_1, \ldots, \tau_n\}$ *if the following predicate holds:*

$$\text{scenario}(\Gamma, \mathbf{S}) \overset{def}{=} \exists w \in \mathbb{R}, \forall S_i : (S_i + w) \mod T_i = 0 \land \max(\mathbf{S}) - S_i < T_i$$

Both conditions defined by the above predicate mean that: (a) $\mathbf{S}$ is a tuple of tasks release times; and (b) only the closest jobs, released before the last released job, are considered. As illustration, consider Example 4.1.
Example 4.1. *Consider* $\Gamma = \{\tau_1, \tau_2, \tau_3\}$ *a set of three periodic tasks and let their periods be* $\mathbf{T} = (10, 15, 20)$ *and define* $h = \text{lcm}(T_1, T_2, T_3) = 60$ *the hyperperiod of this task set. Figure 5 shows an Earliest Deadline First schedule for this task set.*

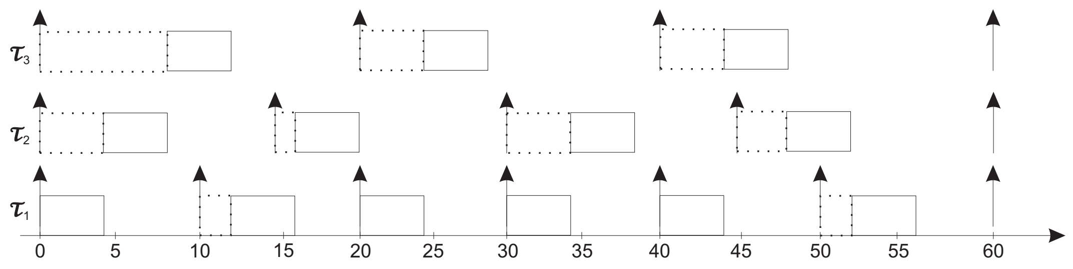


Figure 5. Illustration of an EDF schedule for Example 4.1

Notice that there are h/T_i simulation scenarios for task τ_i since these are the number of its jobs released within the system hyperperiod. Considering Example 4.1, tuples $(0, 0, 0)$, $(20, 15, 20)$ and $(40, 30, 40)$ are simulation scenarios according to Definition 4.1. However, tuple $\mathbf{S} = (40, 15, 40)$, say, is not. In this example, although S_i is a possible release time of τ_i $(i = 1, 2, 3)$, the release time of τ_2 should be 30 instead of 15 to make $\mathbf{S}$ a simulation scenario for this task set example. Simulation scenarios, its properties and generation procedures have been recently discussed [Lima and Nascimento 2009]. Here we assume that procedures to generate non-biased sets of simulation scenarios are available.

As long as simulation scenarios have been generated the simulation engine computes the fault resilience metric for each scenario. Considering that $f_i^{\mathbf{S}}$ is the minimum number of errors that make a task τ_i unschedulable in a given simulation scenario $\mathbf{S}$, the effort value is defined as follows:

Definition 4.2. *Let* $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ *be a task set and consider* $\mathbf{S}$ *a simulation scenario of* Γ*. The effort* $\mathbf{E}_i$ *made by the error generator to make a task* $\tau_i \in \Gamma$ *unschedulable in* $\mathbf{S}$ *is:*

$$\mathbf{E}_i = \frac{f_i^{\mathbf{S}}}{D_i} \tag{3}$$

Notice that the effort definition is in accordance with Definition 3.1. Also, considering the effort for several simulation scenarios allows one to infer the overall behavior of jobs from fault resilience view point, as stated in Requirement 3.2. Intuitively, the higher the effort $\mathbf{E}_i$ made by the error generator, the higher the resilience of τ_i regarding scenario $\mathbf{S}$. Note that the found values of $\mathbf{E_i}$ can be statistically analyzed. Also, one may be interested in other parameters, such as the effort mean value $\bar{\mathbf{E}}_i$ or the minimum effort $\mathbf{E}_i^{\min}$ necessary to make a given system task unschedulable.

For complex systems, determining $\mathbf{E}_i$ or any function of it may be too time consuming since the total number of simulation scenarios is a function of the hyperperiod of the task set. This simulation environment is able to study $\mathbf{E}_i$ through sampling. The following section briefly presents this procedure through an illustrative example.

Assume that one wishes to evaluate the effort for a task τ_i in $\Gamma = \{\tau_1, \ldots, \tau_n\}$, considering a specific simulation scenario $\mathbf{S} = (S_1, \ldots, S_n)$. The job of this task, released at S_i, namely J_i, is called hereafter the analyzed job. Figure 6 sketches the simulation process and will be used for illustration purposes. Scenario $\mathbf{S}$ and a previous scenario $\mathbf{S}'$ are indicated in the gray area of the figure. Release times are indicated by the vertical arrows. The first release time of jobs in $\mathbf{S}$ whose priorities are greater than that of the priority of J_i is denoted $r \leq S_i$ in the figure. These jobs must be considered when analyzing the effects of errors in the execution of J_i, since they can interfere in J_i.

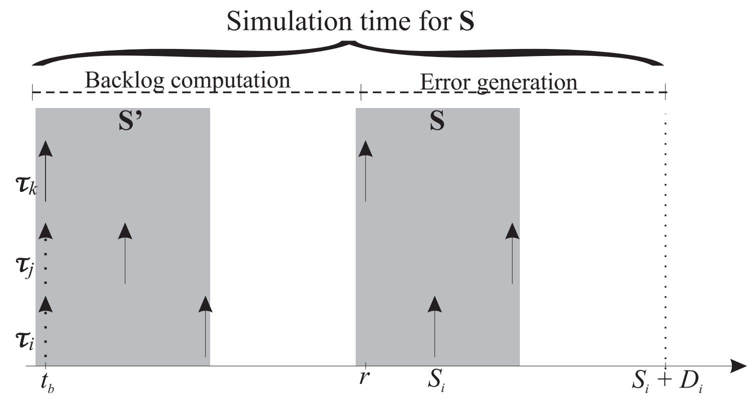


Figure 6. Two-step simulation procedure used by the Simulation Engine

The simulation of the system regarding $\mathbf{S}$ involves two problems: (a) determining the execution backlog at r, which is related to jobs released before r; and (b) generating the minimum number of errors from r onwards so that the analyzed job misses its deadline. Nonetheless, exact solutions to problems (a) and (b) may be computationally too expensive. Thus, our approach to solving them is to derive an upper bound for (a) and a lower bound for (b) so that the effort of the error generator is not overestimated. The simulation procedure has two steps, as illustrated in Figure 6.

The approach to estimating an upper bound on the backlog at r is based on (a) going back to a previous scenario $\mathbf{S}'$ and (b) forcing the release time of all tasks in Γ be at time t_b, as indicated by the dotted arrows in Figure 6. Then the remainder task execution time after simulating the system within $[t_b, r)$ should give the desired upper bound. It is worth mentioning that ideally $\mathbf{S}'$ should be a scenario which gives a good trade-off between simulation time and backlog estimation. In the context of this work, though, we follow a simple approach to computing $\mathbf{S}'$, which is backtracking to the closest simulation scenario.

The simulation during $[r, S_i + D_i)$ is carried out with the error generator active. The strategy is to generate errors in the job which causes the highest interference in the analyzed job. Since the goal is to estimate a lower bound on the minimum number of generated errors that make the analyzed job miss its deadline, the faulty jobs are allowed to execute beyond its deadline. In other words, the optimization problem of determining which jobs fail during simulation is circumvented. According to this approach the found number of errors is guaranteed not to be overestimated but can be underestimated.

As mentioned before, when simulating a scenario $\mathbf{S}$ for a given analyzed job J_i, the error generator must not let J_i meet its deadline. Hence, every time t at which J_i would successfully finish its execution (i.e $t \leq S_i + D_i$), an error must be generated in some job J_j. The recovery of this faulty job must be such that it maximizes the interference in the execution of J_i, so that $f_i^{\mathbf{S}}$ is actually the minimum number of errors necessary to make a task miss its deadline in scenario $\mathbf{S}$. To do so, the error generator considers either jobs released before or after S_i to generate an error.

To make things clear, consider $\mathbf{S} = (50, 45, 40)$ a simulation scenario for Example 4.1. In this scenario let us analyze the job of τ_3 released at $t = 40$. For such a scenario, $\mathbf{S}' = (40, 30, 40)$ and the simulation time interval is $[30; 60)$. Figure 7 shows the whole scheduling for $\mathbf{S}$, including the backlog computation and error generation.

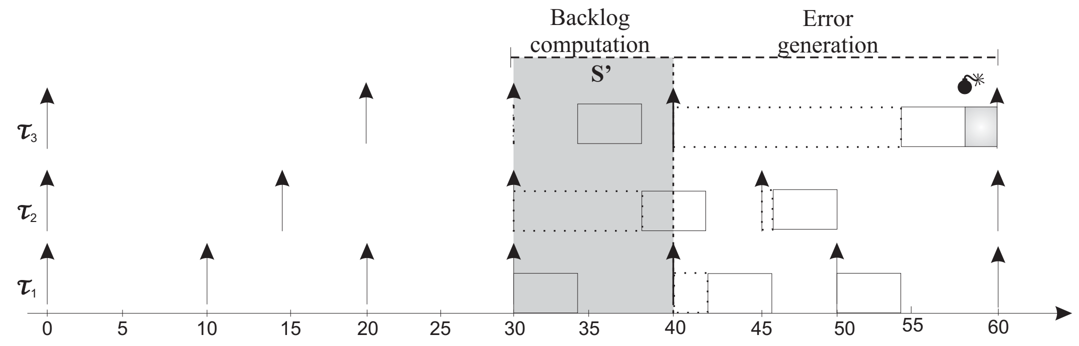


Figure 7. Illustrative simulation Example considering Example 4.1

Observe that, in this case, system simulation starts at $t_b = 30$. At this time, an instance of τ_3 is artificially included (dotted arrow in Figure 7). As mentioned before, this is done for deriving an upper bound on the backlog. Notice that at $t = 40$, a backlog job from previous scenario (from task τ_2) is still executing. It is worth mentioning that in the actual schedule shown in Figure 5 the backlog is null.

At time instant $t = 40$ the analyzed job is released and starts executing at $t = 54$ due to the interference of higher priority jobs. When the error generator realizes that this job is going to finish its execution successfully, it generates an error so that this job miss its deadline, making the system unschedulable. The choice of the faulty job is done so that the error generator effort is minimized. Hence, any active job in $[40, 60)$ can be chosen as the faulty job provided that it causes the highest interference in the analyzed job. It is important to notice that during the time interval $[40, 60)$, if any other job released at this time interval misses its deadline, the system is still considered schedulable. Only the timeliness of the analyzed job (J_3) is observed. According to Definition 4.2, the effort value for $\mathbf{S}$ is given by $\mathbf{E}_3 = \frac{f_3^{\mathbf{S}}}{D_3} = \frac{1}{20} = 0.05$.

In the following section we present some simulation results showing how the derived fault resilience metric can be used to compare different systems.

5. Simulation Results

In this section we present two examples to illustrate how the simulation-based approach works. First, we consider the same fixed-priority system used by other researchers [Burns et al. 1996] to compare the results obtained by our analysis with the ones obtained previously.

Example 5.1. *Let* $\Gamma = \{\tau_1, \ldots, \tau_4\}$ *be a task set of four independent periodic tasks scheduled by Rate Monotonic. Task attributes are* $\mathbf{C} = (30, 35, 25, 30)$*,* $\mathbf{T} = (100, 175, 200, 300)$ *and* $\mathbf{D} = \mathbf{T}$.

Since this task set is considerably small, it is possible to generate all possible scenarios for any task $\tau_i \in \Gamma$. In this case, we focus on task τ_4. When all simulation scenarios of τ_4 are submitted to the simulation engine, the number of errors that make τ_4

miss its deadline can be 2, 3 or 4 with probabilities 0.282, 0.571 and 0.143, respectively. It is worth mentioning that the analysis by Punnekkat *et al.* [Burns et al. 1996] gives $\min(f_4) = 2$ but the distribution of f_4 gives more information about the system fault resilience. Indeed, based on this distribution, we can compute the effort whose mean for this example is $\bar{\mathbf{E}}_i = 0.0044$.

The derived metric can also be used to compare different scheduling policies from fault resilience view point. In order to carry out this kind of evaluation, we randomly generated ten task sets, each one with ten tasks. Both C_i and $\bar{C}_i$ were set to one time unit. Task periods were randomly generated so that processor utilization was bounded in $[0.6; 0.8)$. The scheduling for each generated task set was simulated considering both RM and EDF. We computed the mean effort $\bar{\mathbf{E}}_i$ that makes their tasks τ_i miss their deadlines, $i = 1, 2, \ldots 10$.

Using a procedure to randomly generate simulation scenarios [Lima and Nascimento 2009], we took a sample of simulation scenarios for each analyzed task. In order to determine the sample size, standard statistical sampling procedures were used [Triola 2008]. The maximum assumed error was $|\bar{E}_i^* - \bar{E}_i| = 5\text{x}10^{-3}$, where $\bar{E}_i^*$ and $\bar{E}_i$ stand for the mean values for the effort related to the sample and to the population (not necessarily known), respectively. The result is presented in Table 1, where CI stands for the confidence interval.

Table 1. $\bar{E}_i$ with 95% of confidence

	RM		EDF	
i	$\bar{\mathbf{E}}_i^*$	CI_i	$\bar{\mathbf{E}}_i^*$	CI_i
1	1.000	[1.000,1.000]	0.998	[0.995,1.000]
2	0.613	[0.470,0.750]	0.675	[0.601,0.780]
3	0.500	[0.450,0.630]	0.576	[0.412,0.639]
4	0.413	[0.378,0.560]	0.487	[0.333,0.605]
5	0.397	[0.335,0.516]	0.432	[0.276,0.564]
6	0.360	[0.295,0.490]	0.359	[0.275,0.507]
7	0.278	[0.210,0.378]	0.321	[0.247,0.428]
8	0.235	[0.196,0.356]	0.280	[0.187,0.399]
9	0.221	[0.190,0.300]	0.277	[0.198,0.399]
10	0.110	[0.089,0.200]	0.169	[0.215,0.212]

As can be seen, EDF has a better overall performance in terms of fault resilience. Although this behavior was expected due to the optimality of EDF in terms of schedulability [Liu 2000, Aydin 2007], it is important to emphasize that now the difference is being measured.

It is worth mentioning that τ_1 has approximately the same fault resilience for both schedulers. Since no other task interferes in the execution of τ_1 according to RM and only in a few simulation scenarios there are other tasks with priority greater than the priority of τ_1 by EDF, this behavior was also expected. On the other hand, for all other tasks, EDF is visibly superior to RM in terms of fault resilience. Obviously, we are not considering here problems such as possible overloads caused by admission of recovery actions, which could make EDF degrade. Nonetheless, the goal of the analysis is to point out to what extent the system support errors and is not on evaluating overload conditions. Hence, we are not concerned with scheduling anomalies.

6. Conclusions

In this paper we presented a metric which aims at measuring fault resilience of real-time systems. Such systems are characterized by their need to meet both logical and timing application requirements, even in the presence of faults. However, since errors are random events and cannot be predicted, most approaches assume a worst-case error pattern to analyze whether the system is timely under fault scenarios. We intend to measure the fault resilience of real-time systems, an issue not addressed before. Our analysis is not based on a specific error pattern and is independent of the system model. Indeed, we showed that our analysis can be used to compare different scheduling policies. Moreover, our analysis can also be used to subsidize system designers for choosing the scheduling mechanism that best suit their system.

Some aspects must be further investigated. For example, considering less strict task models, where not all tasks are periodic, may be interesting. Also, taking spacial redundancy into account would enrich the analysis described here. In any case, the concept of simulation scenario and/or the simulation procedure described here must be extended. We believe that the results presented here are a step towards these challenging research issues.

References

[Avizienis et al. 2004] Avizienis, A., Laprie, J.-C., Landwehr, C., and Randell, B. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

[Aydin 2007] Aydin, H. (2007). Exact fault-sensitive feasibility analysis of real-time tasks. *IEEE Transactions on Computers*, 56(10):1372 – 1386.

[Broster and Burns 2004] Broster, I. and Burns, A. (2004). Random Arrivals in Fixed Priority Analysis. In *Proc. of the 1st International Workshop on Probabilistic Analysis Techniques for Real-time and Embedded Systems*.

[Burns et al. 2003] Burns, A., Bernat, G., and Broster, I. (2003). A Probabilistic Framework for Schedulability Analysis. In *Proc. of the Third International Conference on Embedded Software*, pages 1 – 15.

[Burns et al. 1996] Burns, A., Davis, R., and Punnekkat, S. (1996). Feasibility Analysis of Fault-Tolerant Real-Time Task Sets. In *Proc. of the 8th Euromicro Conference on Real-Time Systems*, pages 29 – 33.

[Burns et al. 1999] Burns, A., Punnekkat, S., Strigini, L., and Wright, D. R. (1999). Probabilistic Scheduling Guarantees for Fault-Tolerant Real-Time Systems. In *Proc. of the International Working Conference on Dependable Computing for Critical Applications*, pages 361 – 378.

[Burns and Wellings 2001] Burns, A. and Wellings, A. J. (2001). *Real-Time Systems and Programming Languages*. Addison-Wesley, 3rd edition.

[Ghosh et al. 1995] Ghosh, S., Melhem, R., and Mossé, D. (1995). Enhancing Real-Time Schedules to Tolerate Transient Faults. In *Proc. of the 16th IEEE Real-time Systems Symposium*, pages 120–129.

[Ghosh et al. 1998] Ghosh, S., Melhem, R., Mossé, D., and Sarma, J. S. (1998). Fault-tolerant rate monotonic scheduling. *Real-Time Systems*, 15(2):149–181.

[Han et al. 2003] Han, C., Shin, K., and Wu, J. (2003). A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults. *IEEE Transactions on Computers*, 52(3):362–372.

[Huselius et al. 2007] Huselius, J., Kraft, J., Hansson, H., and Punnekkat, S. (2007). Evaluating the Quality of Models Extracted from Embedded Real-Time Software. In *Proc. of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 577–585.

[Liberato et al. 2000] Liberato, F., Melhem, R., and Mossé, D. (2000). Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems. *IEEE Transactions on Computers*, 49(9):906–914.

[Lima and Burns 2003] Lima, G. and Burns, A. (2003). An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems. *IEEE Transactions on Computers*, 52(10):1332–1346.

[Lima and Burns 2005] Lima, G. and Burns, A. (2005). Scheduling Fixed-Priority Hard Real-Time Tasks in the Presence of Faults. In *Proc. of the 2nd Latin-American Symposium on Dependable Computing*, volume LNCS 3747, pages 154–173. Springer-Verlag.

[Lima and Nascimento 2009] Lima, G. and Nascimento, F. (2009). Simulation Scenarios: a Means of Deriving Fault Resilience for Real-Time Systems. In *Proc. of the 11th Workshop on Real-Time and Embedded Systems*.

[Liu and Layland 1973] Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61.

[Liu 2000] Liu, J. W. S. W. (2000). *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Pandya and Malek 1998] Pandya, M. and Malek, M. (1998). Minimum achievable utilization for fault-tolerant processing of periodic tasks. *IEEE Transactions on Computers*, 47(10):1102–1112.

[Pereira et al. 2004] Pereira, N., Tovar, E., Batista, B., Pinho, L. M., and Broster, I. (2004). A few what-ifs on using statistical analysis of stochastic simulation runs to extract timeliness properties. In *1st International Workshop on Probabilistic Analysis Techniques for Real-time and Embedded Systems (PARTES '2004)*, Pisa, Italy.

[Triola 2008] Triola, M. F. (2008). *Elementary Statistics*. Pearson.

[Wall et al. 2003] Wall, A., Andersson, J., and Norstrom, C. (2003). Probabilistic simulation-based analysis of complex real-time systems. In *Proc. of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 257–266.